

◇ Course Objectives ◇

This course is concerned with the formal development of software.

It is supported by B4Free and ProB, which support the method taught in this course.

You should come away from this course with an understanding and working knowledge of the B approach to the formal development of software systems. This includes an understanding of the following:

- ◇ how to write abstract specifications
- ◇ refining to code
- ◇ the relationship between specifications and code
- ◇ what is required for verification

◇ Course overview ◇

The course aims to cover the following topics:

- ◇ Abstract Machine Notation (AMN)
- ◇ Machines
- ◇ Refinement
- ◇ AMN Code
- ◇ Library and BASE Machines
- ◇ Code generation and implementation
- ◇ Verification
- ◇ Loops
- ◇ Structuring mechanisms

◇ A Partial History ◇

The B method synthesises many approaches to formal methods.

The Z Notation	Abrial
pre and post conditions	Jones
Guarded commands	Dijkstra
Data Refinement	He Hoare Sanders
Refinement Calculus	Morgan
Stepwise Refinement	Gries

◇ Literature ◇

Recommended reading

- ◇ The B Book: Assigning Programs to Meanings — J-R Abrial
- ◇ Specification in B: an introduction using the B-Toolkit — Lano and Haughton
- ◇ Software Engineering with B — J.B. Wordsworth
- ◇ www-scm.tees.ac.uk/bresource/b.html

History:

- ◇ An axiomatic basis for computer programming — C.A.R. Hoare, CACM 1969
- ◇ A discipline of programming — E.W. Dijkstra, Springer-Verlag
- ◇ The science of programming — D. Gries, Springer-Verlag

◇ The B-Method ◇

This course will be based around one particular formal method: the B-Method.

The B-Method incorporates and synthesises many ideas from many approaches to formal methods. It is motivated by the need to support all stages of the software lifecycle in a uniform and formal way.

Abstract Machine Notation is a wide spectrum pseudo-programming language which allows the expression of both abstract specifications and implementation level code.

The B-Method provides an approach to formal development using AMN.

The B-Toolkit and *AterlierB/B4Free* are CASE tools which provides support for the B-Method.

The guiding principle is *simplicity*.

◇ Software Engineering ◇

1. Requirements
 2. Specification
 3. Design
 4. Implementation
 5. Verification
 6. Maintenance
- + prototyping

◇ The B Toolkit* & B4Free ◇

- ◇ syntax and type checking
- ◇ animation (ProB)
- ◇ prototyping
- ◇ generating code (new version of B4Free)
- ◇ managing large developments
- ◇ generating proof obligations
- ◇ proving
- ◇ code module library*
- ◇ translation to C (new version of B4Free)
- ◇ remake facilities*

◇ Abstract Machines ◇

The building block of the B-method is the concept of an abstract **machine**. It is similar to an abstract object.

An abstract machine has

- ◇ a name
- ◇ local *state*
- ◇ a collection of *operations* which may access and update the state.

An abstract machine is a *specification* of what the object will provide.

In other words, it describes *what* the machine will provide, but not *how* it will provide it.

It will be written in terms of how the user should think about the machine.

◇ State ◇

The *state* of the machine should correspond to how the machine is to be understood. It need not correspond to what can be implemented directly on a computer.

State variables can have whatever type is most appropriate to the specification.

For example, a machine which keeps track of a set of numbers can use a variable

numset

which contains the set of numbers.

[Note: In the B-Toolkit, variable names must be at least two characters long. This is why they are often called *aa* or *vv*. This is not required for B4Free.]

◇ Type information ◇

Types are particular sets.

Each state variable must be accompanied by information concerning its type.

Typing information about the state variables is contained in the machine's INVARIANT.

Type information is of the form $ee : SS$ or $ee < : SS$.

The first states that ee is a member of the type SS , or equivalently that ee has type SS .

The second states that ee is a subset of SS , or equivalently that it has type $\mathbb{P} SS = \text{POW}(SS)$

`numset < : NAT`

`numset : POW(NAT)`

◇ Machine readability ◇

The B-Toolkit requires ASCII versions of specifications.

Sets:

Sets	ASCII	meaning
$S \cup T$	$S \ \vee \ T$	union
$S \cap T$	$S \ /\ \ T$	intersection
$e \in S$	$e : S$	member of
$e \notin S$	$e / : S$	not member of
$S \subseteq T$	$S < : T$	subset
$S \setminus T$	$S - T$	set subtraction
$\mathbb{P} S$	$\text{POW}(S)$	power set
$ S $	$\text{card}(S)$	size
\mathbb{N}	NATURAL	naturals
\mathbb{N}	NAT	implementable naturals
\mathbb{N}_1	NAT1	positive numbers

◇ Machine readability II ◇

Logic:

logic	ASCII	meaning
$P \vee Q$	P or Q	or
$P \wedge Q$	P & Q	and
$\neg P$	not(P)	negation
$P \Rightarrow Q$	P => Q	implication
$\forall x : T \bullet P$!x.(x:T => P)	for all
$\exists x : T \bullet P$	# x.(x:T & P)	there exists

All notation is available in the B-Toolkit online help.

◇ Examples ◇

A 'paper round manager' keeps track of houses that receive deliveries.

It uses a state variable `houseset`.

It might be concerned with the following claims:

- ◇ `houseset <: NAT`
- ◇ `houseset : POW(NAT1)`
- ◇ `houseset <: 1..163`
- ◇ `(houseset /= {}) => (3:houseset)`
- ◇ `not(houseset = { })`
- ◇ `hh : houseset`
- ◇ `houseset - {hh1, hh2} = {}`
- ◇ `(card(houseset) >= 40) or (139/:houseset)`
- ◇ `!hh. (hh:houseset => hh < 163)`

What do each of these say?

◇ Operations ◇

In general, specifications of operations require the following information:

- ◇ Name
- ◇ Input parameters
- ◇ Output parameters
- ◇ **requires** (restrictions on parameters)
- ◇ **modifies** (inputs and global variables that may be modified)
- ◇ **effects** (the behaviour of the process)

Example—the paper round manager:

Specify an update operation to add a new house to the round.

Specify a query operation to ask how many houses are on the round.

◇ Operation interfaces ◇

The inputs ii and outputs oo associated with an operation op are declared in B as follows:

```
oo <-- op(ii)
```

Examples:

```
add(new)
```

has no output parameters, and one input parameter: new

```
ans <-- number
```

has one output parameter ans , and no input parameter

In general, ii and oo can be *lists* of input and output variables. They should all be distinct.

◇ Preconditions ◇

An operation is described as

PRE P

THEN S

END

P is the *precondition* on the operation. It describes restrictions on the parameters and on the state of the machine. The operation should only be called when the precondition is true.

The precondition must give the *type* of all input variables. It can also include other constraints.

Examples:

new:NAT1

new:NAT1 & new < 163

◇ Operation body ◇

The body of the operation S describes its effect. In other words, it describes what the operation does.

It must describe how the machine's state is updated, and the output to be provided.

This is described by an abstract assignment.

The state values may optionally be assigned (if they are not, the operation does not alter them)

Any output variables must be assigned.

Examples:

```
houseset := houseset \ / {new}
```

```
ans := card(houseset)
```

In the second case, `houseset` remains unchanged.

◇ Operation specifications ◇

Putting all the components together:

```
add(new) =  
PRE  
new:NAT1 &  
new /: houseset  
THEN  
houseset := houseset \ / {new}  
END;
```

```
ans <-- number =  
BEGIN  
ans := card(houseset)  
END
```

Operations listed in a machine description are separated by semi-colons.

◇ The whole machine ◇

```
MACHINE Paper_Round
VARIABLES houseset
INVARIANT houseset:POW(NAT1)
INITIALISATION houseset := {}
OPERATIONS
add(new) =
PRE new:NAT1 & new/:houseset
THEN houseset := houseset \ / {new}
END;
ans <-- number =
BEGIN ans := card(houseset)
END
END
```

A machine contains a number of clauses.

◇ Other aspects ◇

- ◇ **Name:** A machine must have a name
- ◇ **Initialisation:** The state must be initialised, by means of an assignment to the state variables.
- ◇ **Constants** can also be introduced into a machine description.

◇ Introduction ◇

Last week we introduced the concept of Abstract Machine.

MACHINE name

VARIABLES vv

INVARIANT I

INITIALISATION T

OPERATIONS

o1 \leftarrow op1(i1) =

PRE

P1

THEN

S1

END

END

Today's lecture will be looking at operations and invariants in more detail.

◇ Further AMN ◇

Operations describe a one-step change of state and assignment of output. Specifications need only be concerned with the relationship between initial and final state, and intermediate states are not appropriate to specification (so no loops or sequencing).

The body of an operation (and the initialisation clause) is specified using the pseudo-programming language AMN. This lecture will cover some other parts of AMN, as follows:

- ◇ assignment
- ◇ multiple assignment
- ◇ conditional: if-then-else
- ◇ skip: no-op; does not change the state.
- ◇ BEGIN S END: delimits S
- ◇ parallel

◇ Assignment ◇

An assignment has the form

$$x := E$$

It calculates the value of the expression E , and then overwrites the value of variable x so it now contains this value.

A multiple assignment has the form

$$x, \dots, y := E, \dots, F$$

All of the expressions E, \dots, F are first of all evaluated, and then the result of each is *simultaneously* assigned to the corresponding variable, overwriting its previous contents.

Examples:

$$x, y := y, x$$
$$\text{houseset}, \text{num} :=$$
$$\text{houseset} \setminus \{\text{new}\}, \text{card}(\text{houseset})$$

◇ Conditional ◇

A conditional has the form

```
IF B THEN S ELSE T END
```

ELSIF clauses are also possible

The ELSE clause is optional. If it is omitted, and none of the conditions are true, then the state remains unchanged.

```
IF x > y THEN x,y := y,x END
```

```
IF x = y
  THEN maj := x
ELSIF x = z
  THEN maj := x
ELSE maj := y
END
```

```
IF x = y
THEN maj := x
ELSE maj := z
END
```

◇ CASE statement ◇

```
CASE E OF  
  EITHER 1 THEN S  
  OR m THEN T  
  . . .  
  ELSE V  
END  
END
```

The ELSE is optional. If it's not there, then the state remains unchanged if none of the cases fit.

◇ Parallel ◇

Statements can also be specified in parallel:

$S \parallel T$

The variables updated by S and by T must be distinct.

Examples:

$x := 4 \parallel y := 7$ is equivalent to
 $x, y := 4, 7$

$x := y \parallel y, z := x, 7$ swaps x and y , and assigns 7 to z

The components of a parallel composition can be any statement, not only assignment:

$x := 4 \parallel \text{IF } y > x \text{ THEN } y, z := x - 1, y \text{ END}$

◇ Invariants again ◇

Invariants provide type information about the variables of the machine.

They can also capture other conditions that *must* hold on the state: on particular variables and on relationships between them.

For example, the paper round manager might require that no more than 75 houses can be accepted. In this case, the invariant would include the clause

```
card(houseset) <= 75
```

Invariants contain consistency conditions capturing the specifier's understanding about the information contained in the machine. If they are violated during an execution of the machine, then something has gone wrong.

The invariant is sometimes called the *static specification* of the machine: it describes something that must be true of every state the machine can reach.

◇ Operations ◇

The operations describe how the machine can change state. They are sometimes called the *dynamic specification* of the machine, because they specify how it will run.

Operations and invariants are tightly coupled: invariants must always be preserved by all of the operations, (provided they are called within their precondition).

Question: do the operations of Paper_Round preserve the new invariant:

```
houseset : POW(NAT1)
& card(houseset) <= 75
```

◇ Initialisation ◇

The initialisation of the machine must establish the invariant in the first place. This means that the machine starts off in a correct state.

If all of the operations preserve the invariant, then the machine can never reach an incorrect state.

◇ Example ◇

Now let's add a second state variable to the `Paper_Round` machine, to keep track of the households that have magazines delivered.

`magazines` is added to the `VARIABLES` clause.

`magazines : POW(NAT1)` is added to the `INVARIANT` clause as type information.

We also expect that magazines will only be delivered to people registered on the round, though not necessarily all of them. This means that

`magazines <: houseset`

should also be part of the invariant.

The invariant can describe consistency conditions between different parts of the state.

The initialisation clause will be

`houseset := {} || magazine := {}`

◇ Example ctd ◇

An operation to add a house to the magazine set would be specified as follows:

```
add_magazine(new) =  
PRE  
???  
THEN  
magazine := magazine \ / {new}  
END
```

Question:

What precondition will ensure that this operation preserves the invariant

```
houseset : POW(NAT1)  
& magazine : POW(NAT1)  
& card(houseset) <= 75  
& magazine <: houseset
```

◇ Sets ◇

Suppose we wish to construct an abstract machine to keep track of the courses students are registered on. We would wish to introduce a STUDENT type, a COURSE type, and a GRADE type in order to talk about students and courses.

At the specification level, we are not concerned with how these will be implemented, or even what they consist of.

In other specification languages such as Z, such sets are introduced as

[STUDENT, COURSE, GRADE]

These are then treated as types in the same way as NAT.

◇ The SETS clause ◇

One way of introducing sets is by means of the SETS clause in a machine description.

SETS

STUDENT; COURSE; GRADE

Sets listed in the SETS clause are separated by semicolons.

Information about sets in the SETS clause can come from three places:

- ◇ The set can be *enumerated*, e.g. GRADE = {dist, pass, fail}
- ◇ Properties of the set can be described in the PROPERTIES clause, e.g.
card(STUDENT) > 7
- ◇ If the set is not defined in the machine, then it is said to be *deferred*. In this case, the information will be provided at a later stage during implementation.

◇ Constants and properties ◇

The `CONSTANTS` clause lists all of the constants of the machine. Particular elements of deferred types may be listed here, e.g. `cs381`, `cs382`

The `PROPERTIES` clause must contain type information about all of the constants of the machine. Such types must be either standard types or sets that have been introduced.

```
cs381:COURSE & cs382:COURSE
```

The `PROPERTIES` clause may also contain other information about the sets and constants.

```
cs381/=cs382 & card(STUDENT) < 800
```

◇ Parameters ◇

Genericity of machines is provided by parameters.

Sets and constants can be provided as parameters of the machine. Sets are written in upper case, and constants in lower case.

Register(GRADE, top, maxreg)

The sets that are passed can be used as types within the machine.

Any constraints about these parameters is provided in a CONSTRAINTS clause. This clause must contain type information about all of the constants.

CONSTRAINTS

maxreg : NAT1 & top : GRADE

It cannot constrain the *type* of the sets being passed as parameters.

CONSTRAINTS

card(GRADE) >= 2 & maxreg : NAT1

◇ Example ◇

A machine description might start as follows:

```
MACHINE Register(GRADE,top,limit)
```

```
CONSTRAINTS
```

```
  top:GRADE & limit:NAT1  
& limit > 2 & card(GRADE) >=2
```

```
SETS
```

```
  STUDENT; REPORT = { ok, error };  
  COURSE
```

```
CONSTANTS
```

```
  cs381, cs382, maxreg
```

```
PROPERTIES
```

```
  cs381:COURSE & cs382:COURSE  
& cs381 /= cs382 & maxreg:NAT1  
& card(COURSE) <= limit
```

```
...
```

◇ Revision: Relations ◇

[Potter, Sinclair and Till, pp 79–93]

Machine readable notation will be given, and relation notation (where it differs) will be given in brackets.

A relation between a set S and a set T relates elements in S to elements in T . It is simply the set of those pairs (s, t) which are related.

The notation $S \leftrightarrow T$ ($S \leftrightarrow T$) denotes relations between S and T .

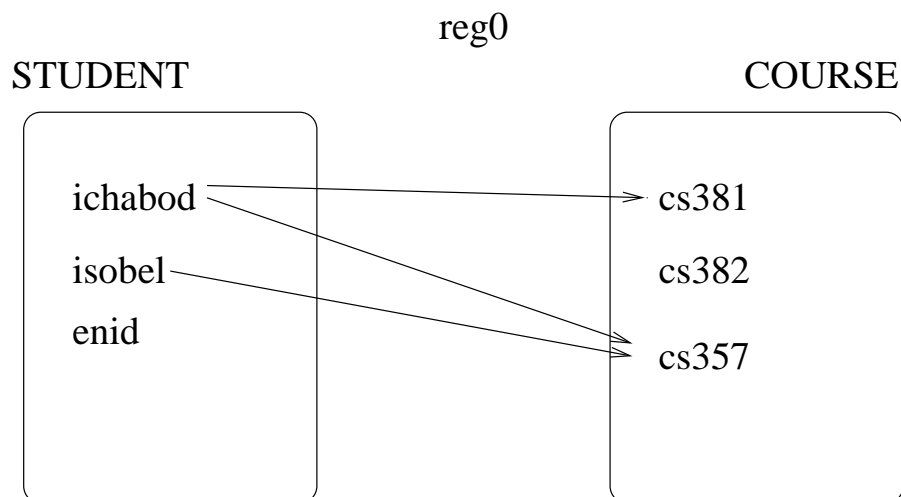
◇ Example ◇

$\text{reg}:\text{STUDENT} \leftrightarrow \text{COURSE}$

states that reg is a relation between STUDENT and COURSE .

Pairs are written using the 'maps to' notation:
 $\text{ichabod} \mapsto \text{cs357}$ ($\text{ichabod} \mapsto \text{cs357}$) is the same as $(\text{ichabod}, \text{cs357})$ (not allowed in ProB and jbedit)

$(\text{ichabod} \mapsto \text{cs357}):\text{reg}$ states that ichabod is related to cs357 in the relation reg .



◇ Extracting information ◇

We're interested in extracting information from the relations our machines maintain.

e.g 'is isobel registered for any courses?', 'how many courses is ichabod registered for?', 'who is registered for CS357?', 'which courses are both ichabod and isobel registered for?'

There are constructs for projecting particular information contained in relations:

- ◇ Extracting the domain and range
- ◇ Domain restriction
- ◇ Range restriction
- ◇ Relational image
- ◇ Relational inverse

◇ Domain and range ◇

The *domain* of a relation $R : S \leftrightarrow T$ is the set of elements of S that R relates to something in T .

It is written $\text{dom}(R)$.

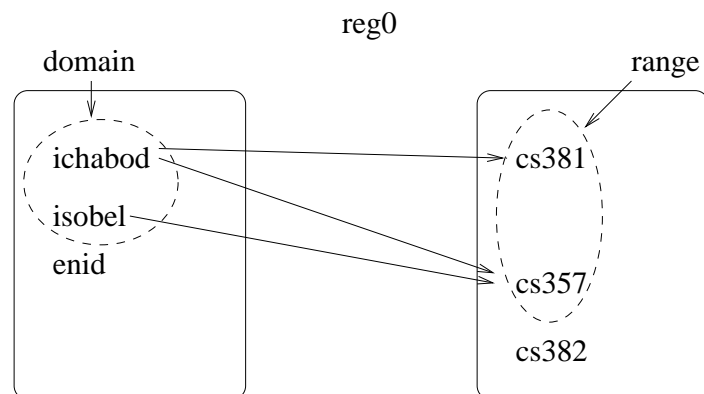
In the relation reg0 , the domain is ichabod and isobel :

$$\text{dom}(\text{reg0}) = \{\text{ichabod}, \text{isobel}\}$$

The *range* is the set of elements of T that are related to something in S .

It is written $\text{ran}(R)$

$$\text{ran}(\text{reg0}) = \{\text{cs357}, \text{cs381}\}$$



◇ Domain restriction ◇

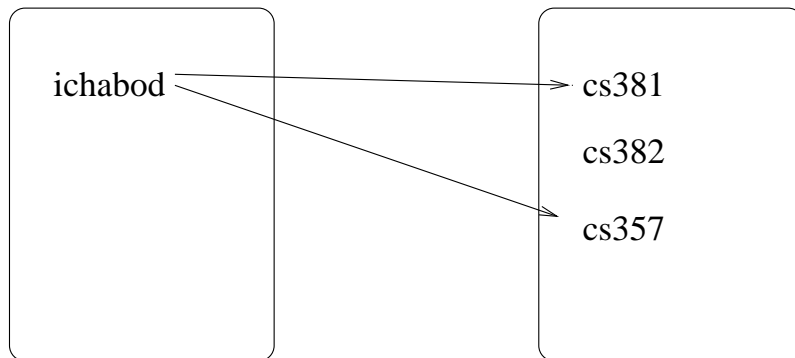
A relation $R : S \leftrightarrow T$ can be projected onto a particular domain $U \subseteq S$.

The result is the pairs in R whose first element is in U . It is written $U \llcorner R$ ($U \triangleleft R$). This is *domain restriction*

The opposite: all pairs apart from those whose first element is in U , is written $U \lll R$ ($U \triangleleft R$). This is *domain anti-restriction*

To consider only the courses that ichabod is taking, the relation `reg0` can be domain restricted to $\{\text{ichabod}\}$

$\{\text{ichabod}\} \llcorner \text{reg0}$



◇ Range restriction ◇

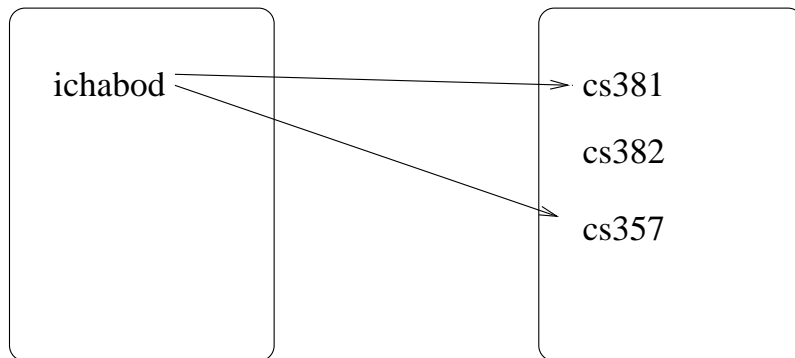
The second component of the relation can also be restricted.

The *range restriction* $R \upharpoonright V (R \triangleright V)$ gives all pairs in R whose second element is in $V \subseteq T$.

The *range anti-restriction* $R \upharpoonright\upharpoonright V (R \triangleright V)$ gives all pairs apart from those whose second element is in V .

To restrict attention only to the course cs357, the relation `reg0` can be range restricted to $\{\text{cs357}\}$

$\{\text{ichabod}\} \upharpoonright \text{reg0}$



◇ Exercise ◇

Let $\text{reg} : \text{STUDENT} \leftrightarrow \text{COURSE}$ be a relation between students and courses that they are registered for.

Express in words what the following correspond to:

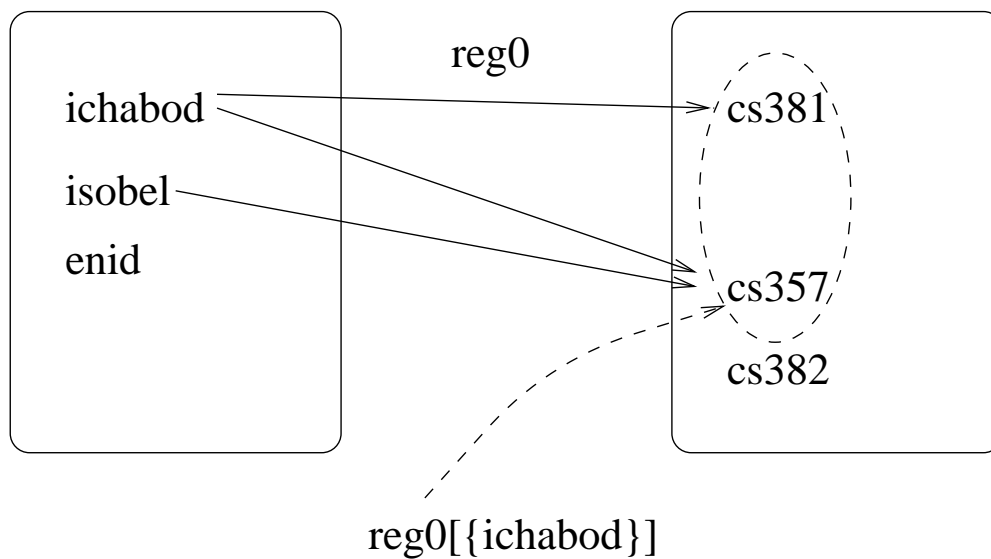
- ◇ $\{ \text{albert}, \text{victoria} \} \ll \text{reg}$
- ◇ $\{ \text{albert}, \text{victoria} \} \ll \ll \text{reg}$
- ◇ $\text{reg} \mid > \{ \text{cs381}, \text{cs382} \}$
- ◇ $\text{dom}(\text{reg} \mid > \{ \text{cs381}, \text{cs382} \})$
- ◇ $\text{ran}(\{ \text{albert}, \text{victoria} \} \ll \ll \text{reg} \mid >> \{ \text{cs381}, \text{cs382} \})$

◇ Relational image ◇

If $U \subseteq S$, then the set of elements in T related to U is called the *relational image* of U .

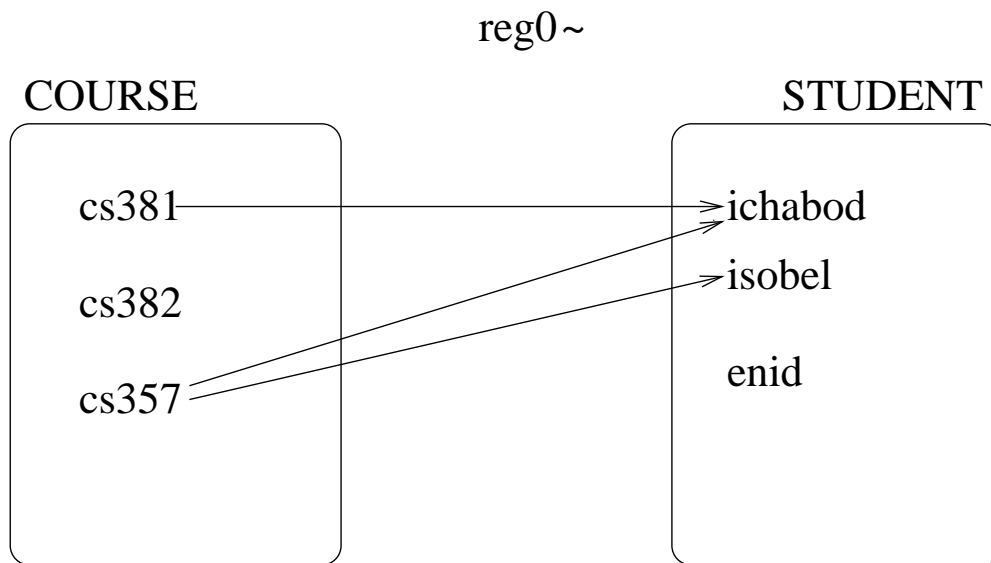
It is written $R[U]$ ($R \upharpoonright U$)

In fact, $R[U]$ is the same as $\text{ran}(U \times |R)$.



◇ Relational inverse ◇

If R is a relation, then its *relational inverse* R^{-1} (R^{-1}) is the relation the other way around: it is of the type $T \leftrightarrow S$, and it maps tt to ss precisely when the original relation mapped ss to tt .



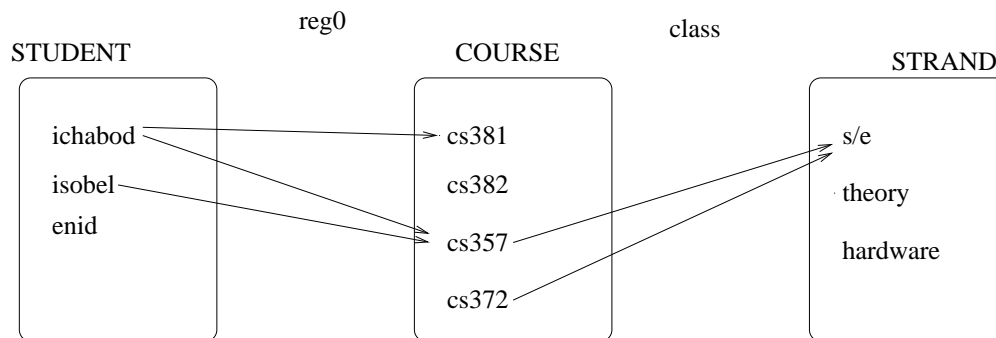
◇ Relational composition ◇

If $R_0 : S \leftrightarrow T$ and $R_1 : T \leftrightarrow W$ are two relations such that the range type of the first is the domain type of the second, then the relations can be composed to give a relation between S and W .

$R_0;R_1 : S \leftrightarrow W$

If $ss \mapsto tt : R_0$ (i.e. R_0 relates ss to tt), and $tt \mapsto ww : R_1$,

then $ss \mapsto ww : R_0;R_1$.



$ichabod \mapsto s/e : reg_0;class$

In other words, ichabod is doing a course classified as s/e.

◇ Exercises ◇

If $\text{reg} : \text{STUDENT} \leftrightarrow \text{COURSE}$ and
 $\text{class} : \text{COURSE} \leftrightarrow \text{STRAND}$
then express in words what the following correspond to:

- ◇ $\text{reg}[\{\text{victoria}\}]$
- ◇ $\text{reg}^{\sim}[\{\text{cs357}\}]$
- ◇ $\text{card}(\text{reg}^{\sim}[\{\text{cs357}\}])$
- ◇ $\text{reg}^{\sim}[\{\text{cs381}\}] \setminus \text{reg}^{\sim}[\{\text{cs382}\}]$
- ◇ $(\text{reg}; \text{class})[\{\text{albert}\}]$
- ◇ $\text{class}^{\sim}[\{\text{theory}, \text{hardware}\}]$
- ◇ $\text{reg}^{\sim}(\text{class}^{\sim}[\{\text{s/e}\}])$

◇ Exercise ◇

Consider a set $PERSON$ and the following relations:

◇ $father : PERSON \leftrightarrow PERSON$

◇ $mother : PERSON \leftrightarrow PERSON$

◇ $brother : PERSON \leftrightarrow PERSON$

◇ $sister : PERSON \leftrightarrow PERSON$

$(Alice, Fred) \in father$ means *Alice's father is Fred*.

Express the following relations in terms of these:

◇ $parent$

◇ $grandmother$

◇ $aunt$

◇ $grandchild$

◇ $cousin$

◇ $half - sibling$ (exactly one parent in common)

◇ $ancestor$

◇ Summary ◇

Relations	ASCII	meaning
$x \mapsto y$	$x \mid\rightarrow y$	x maps to y
$dom(R)$	$dom(R)$	domain of R
$ran(R)$	$ran(R)$	range of R
$U \triangleleft R$	$U < R$	domain restriction
$U \triangleleft\!\!\triangleleft R$	$U << R$	domain anti-restriction
$R \triangleright U$	$R \mid> U$	range restriction
$R \triangleright\!\!\triangleright U$	$R \mid>> U$	range anti-restriction
$R(U)$	$R[U]$	relational image
R^{-1}	$R\sim$	relational inverse
$R0 \circ R1$	$R0 ; R1$	relational composition

◇ Revision: Functions ◇

[Potter, Sinclair and Till, pp 93–110]

Functions are particular kinds of relations. We concentrate on them (and on relations) because they correspond to the kind of structures that we wish to specify in computer systems.

Since functions are special kinds of relation, they are also understood as sets of pairs.

Thus a function f from S to T is simply a set of pairs $ss \mapsto tt$.

What makes it a *function* is that it maps elements of S to at most one element of T .

Example:

`advisor: STUDENT <-> LECTURER` is a function—students have at most one advisor (for the sake of this example!)

`reg: STUDENT <-> COURSE` is not a function—students are registered for many courses.

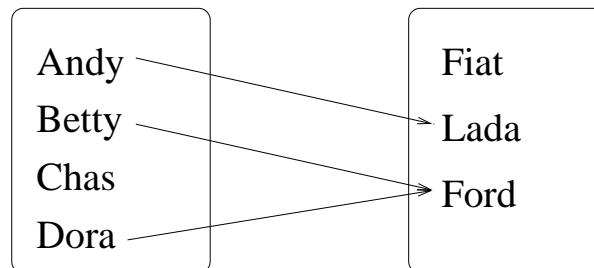
◇ Partial functions ◇

A partial function from S to T is a relation which relates each element in S to *at most one* element in T . Some elements of S might not be related to anything.

$f(s)$ is the value of the function on s .

Notation: $f : S \dashrightarrow T$ ($f : S \twoheadrightarrow T$) states that f is a partial function from S to T . It gives the *type* of f and some additional constraint.

advisor: $\text{STUDENT} \dashrightarrow \text{LECTURER}$ asserts that no student has more than one advisor.



◇ Total functions ◇

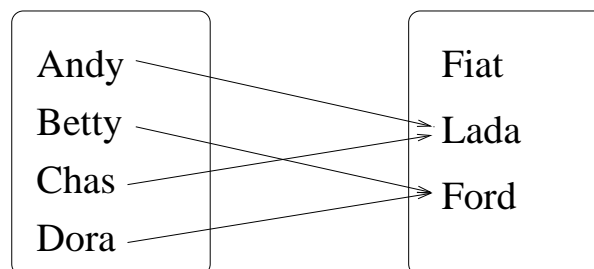
A total function from S to T is a relation which relates each element in S to *exactly one* element in T .

Total functions relate everything in S to something in T .

A total function is a particular kind of partial function.

Notation: $f:S \dashrightarrow T$ ($f : S \rightarrow T$)

`advisor:STUDENT \dashrightarrow LECTURER` states that all students are allocated exactly one lecturer as advisor.



◇ Injective functions ◇

A function is *injective* (or *into*) if it never maps two different elements to the same thing.

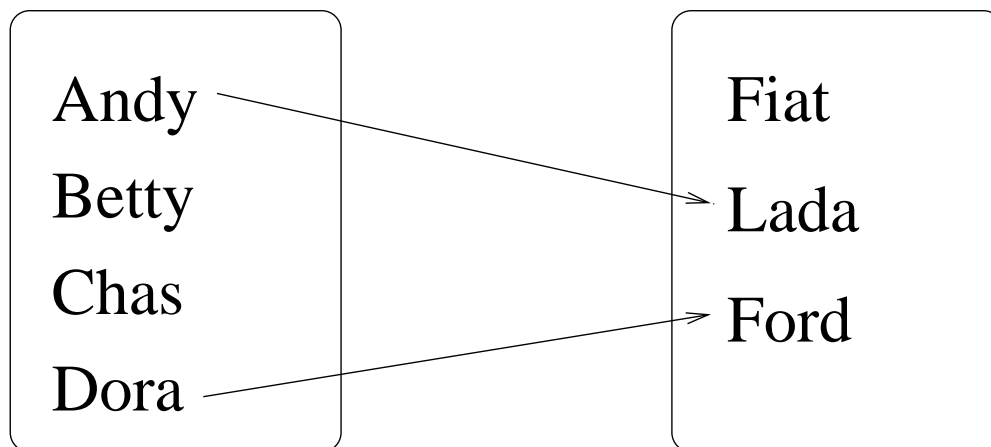
Injective functions can be partial or total.

Notation: $f : S \rightarrow T$ ($f : S \mapsto T$) for total injective functions;

$f : S \rightarrow\!+ T$ ($f : S \mapsto\!+ T$) for partial injective functions.

`examno : STUDENT \rightarrow NAT1`

`s_u_number : STUDENT $\rightarrow\!+$ NAT1`

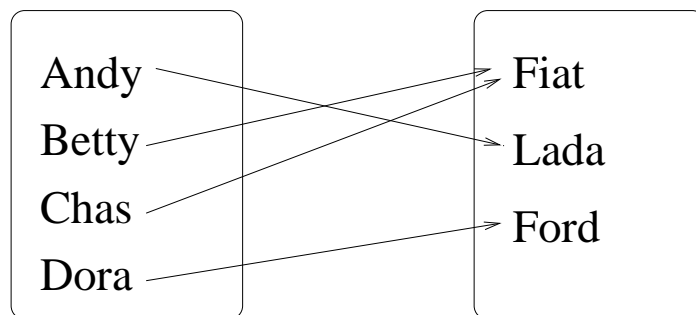


◇ Surjective functions ◇

A function f from S to T is *surjective* (or *onto*) if every element of T is reached from some element of S . In other words, the range of f is the whole of T .

Notation: $f : S \dashrightarrow T$ ($f : S \twoheadrightarrow T$) for total surjective functions;

$f : S \dashrightarrow T$ ($f : S \twoheadrightarrow T$) for partial surjective functions.



Which of these is true in CS?

advisor: STUDENT \dashrightarrow LECTURER

advisor: STUDENT \dashrightarrow LECTURER

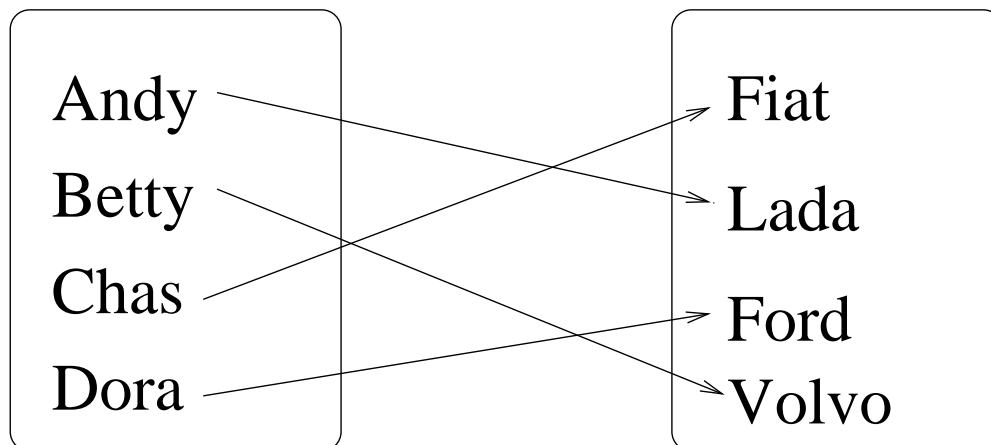
◇ Bijective functions ◇

A function which is total, injective and surjective is called a *bijective* or *one-one* function. Every element in S is mapped to exactly one element in T .

Notation: $f : S \rightarrow T$ ($f : S \mapsto T$) for bijective functions.

Example:

`flag : COUNTRY \rightarrow NATIONAL_FLAG`



◇ Exercise ◇

Which of the following relations are functions?
Of those that are, which are total, which are injective, and which are surjective?

◇ owner : CAR \leftrightarrow PERSON

◇ child : PERSON \leftrightarrow PERSON

◇ child~ : PERSON \leftrightarrow PERSON

◇ mother : PERSON \leftrightarrow PERSON

◇ birthday : PERSON \leftrightarrow DATE

◇ n_i_number : PERSON \leftrightarrow NAT1

◇ f defined by $f(x) = x^2$

◇ f^{-1}

◇ Note in B: $f = \%x.(x:\text{INTEGER} \mid x*x)$
($f = \lambda x.(x:\mathbb{Z} \mid x^2)$)

◇ Summary ◇

Function	ASCII	meaning
$S \mapsto T$	S \mapsto T	partial function
$S \rightarrow T$	S \rightarrow T	total function
$S \mapsto+ T$	S $\mapsto+$ T	partial injection
$S \mapsto- T$	S $\mapsto-$ T	total injection
$S \mapsto\rightarrow T$	S $\mapsto\rightarrow$ T	partial surjection
$S \rightarrow\rightarrow T$	S $\rightarrow\rightarrow$ T	total surjection
$S \mapsto\rightarrow\rightarrow T$	S $\mapsto\rightarrow\rightarrow$ T	(total) bijection

◇ SETS and Types ◇

SETS Students; Lecturers

VARIABLES mensacard

INVARIANT

 mensacard: Students\ /Lecturers \leftrightarrow NAT

What is wrong there?

Solution?

◇ Arrays ◇

Arrays are a common data type in programming.

An array is declared to have a particular type and a given size.

It allows elements of the type to be stored at particular indices, and elements of the array can be indexed with particular indices.

For example, an array a of 6 natural numbers might be represented as follows:

a

3	17	0	5	8	2
---	----	---	---	---	---

For example, $a[2] = 17$

◇ Arrays in B ◇

An array can be viewed simply as a partial function from indices to elements. Accessing the array corresponds to looking up or evaluating the function at that point.

The *type* of an array a of ELEMs is declared as $a : 1..N \rightarrow \text{ELEM}$. An array is then modelled as a simple variable a which has this type.

The set $1..N$ is the set of natural numbers from 1 up to and including N

The fact that a is a partial function means that it does not need to be defined at every point.

The value of a at index i is given by $a(i)$.

◇ Assigning to arrays ◇

The AMN expression for assigning to an array follows programming convention:

$$a(i) := E$$

evaluates the expression E and then writes it to the i th position of a .

This is simply shorthand for updating the function a as follows:

$$a := a \text{ <+ } \{ i \mid \rightarrow E \}$$

[notation: $f \text{ <+ } g$ is *functional over-riding*, giving a function which behaves as g where g is defined, and behaves as f otherwise.]

This means that multiple assignments to the same array are not allowed, e.g.

$a(i), a(j) := 3, 4$ is forbidden.

◇ Assertions about arrays ◇

When arrays are used in machines, they may be constrained in the invariant. In this case they will be treated like any other function.

For example, the **Paper_Round** machine might use an array

$$\text{owing} : 1..163 \rightarrow \text{NAT}$$

to keep track of the money owed by the people who have papers delivered.

In this case, some consistency may be required between `houseset` and `owing`:

`houseset = dom(owing)`, or

$$\text{owing} : \text{houseset} \rightarrow \text{NAT}$$

The second condition states that `owing` must be a total function from `houseset`.

Secondly, no debt may be allowed to become greater than £50. This can be expressed as follows:

$$\text{ran(owing)} <: 0..5000$$

◇ Nondeterminism ◇

All of the AMN introduced so far has been *deterministic*: in other words, there is only one possible final state and outputs for any particular initial state and given inputs.

Determinism in implementations is good, since it assists testing.

Determinism in specification is not necessarily good:

- ◇ sometimes the specifier does not care which course of action is taken;
- ◇ sometimes the specifier is prepared to leave some choices up to the implementor

Forcing the specifier to make decisions which he/she is not in a position to make is inappropriate and unnecessary.

Slogan:

Nondeterminism is underspecification

◇ Example ◇

Suppose that the `Paper_Round` requires an operation `choose`, which selects a house on the round in order to subject them to a special offer.

The specification should not state how this choice is to be made. It requires simply that the output from this operation must be one of the houses in `houseset`.

The deterministic constructs are not appropriate for this:

```
hh <-- choose =  
PRE houseset /= { }  
THEN hh := ???  
END
```

A specification should not be more prescriptive than it needs to be. If something doesn't matter, then the specification should allow it not to matter.

◇ ANY ◇

The assignment style of specifying operations is augmented with ways of making non-deterministic choices.

The ANY statement allows any satisfactory choice to be made, and imposes no further conditions on which should be made. Its syntax is as follows:

```
ANY xx
WHERE P
THEN S
END
```

The way this is executed is as follows: P is a predicate on the xx , which is therefore true for some xx and false for others. It must provide the type of xx . An arbitrary xx is picked for which P is true, and then S (which may contain xx) is executed.

In general, xx can be a list of variables.

◇ choose ◇

```
hh <-- choose =  
PRE houseset /= { }  
THEN ANY xx  
    WHERE xx : houseset  
    THEN hh := xx  
    END  
END
```

This may behave differently each time it is executed. It need not have a deterministic implementation.

◇ Lottery Example ◇

Specifying an operation which makes a selection of numbers for the National Lottery:

```
SS <-- lottery =  
  ANY TT  
  WHERE TT<:1..49 & card(TT)=6  
  THEN SS := TT  
  END
```

Provides a set of six numbers between 1 and 49 as output.

```
lottery2 =  
  ANY bb  
  WHERE bb:1..6-->1..49  
    & card(ran(bb))=6  
  THEN aa := bb  
  END
```

◇ Lottery ctd ◇

```
aa,bb,cc,dd,ee,ff <-- lottery3 =
  ANY aa0,bb0,cc0,dd0,ee0,ff0
  WHERE aa0:1..49 & bb0:1..49
        & cc0:1..49 & dd0:1..49
        & ee0:1..49 & ff0:1..49
        & card({aa,bb,cc,dd,ee,ff})=6
  THEN aa,bb,cc,dd,ee,ff :=
        aa0,bb0,cc0,dd0,ee0,ff0
  END
```

This provides six different outputs between 1 and 49.

◇ CHOICE ◇

A nondeterministic choice between a fixed number of alternatives may also be specified:

```
CHOICE S  
OR T  
...  
OR U  
END
```

This executes by choosing one of the branches S or T ... or U completely arbitrarily, and then executing it.

```
result <-- driving_test =  
CHOICE result := pass  
OR result := fail  
END
```

◇ Nondeterministic assignment ◇

A shorthand notation for a special case of an ANY clause assigns to xx an arbitrary member of the set E :

$xx :: E$

This is sometimes used in initialisation, where it does not matter what the initial state of the machine is.

$nn :: \text{NAT1}$ initialises nn to an arbitrary natural number.

$aa :: 1..N \rightarrow \text{ELEM}$ initialises the array aa to an arbitrary array.

Question: how can $xx :: E$ be written using the ANY construction?

◇ SELECT ◇

Select is like a generalised CASE statement: instead of selecting on specific instances of a particular expression, the selection is on general boolean conditions $P, Q, \dots R$.

```
SELECT P THEN S
WHEN Q THEN T
...
WHEN R THEN U
ELSE V
END
```

If more than one guard is true, then any of the corresponding statements can be executed. This means that nondeterminism can result.

The ELSE clause is optional. If it is not present, then all possible cases should be covered by the other guards $P, Q \dots R$.

◇ SELECT example ◇

```
hh <-- helper =  
SELECT albert : present  
      THEN hh := albert  
      WHEN betty : present  
      THEN hh := betty  
      WHEN clarissa : present  
      THEN hh := clarissa  
      ELSE hh := fido  
      END  
END
```

If more than one of the possibilities are present, then the helper assigned will be nondeterministic.

◇ **B concepts so far** ◇

- ◇ Logical connectives and quantifiers
- ◇ Datatypes: basic SET elements, numbers, booleans, sets, pairs, relations, functions
- ◇ Related operators (card, dom, ...)
- ◇ PARAMETERS+CONSTRAINTS
- ◇ CONSTANTS+PROPERTIES
- ◇ VARIABLES+INVARIANT
- ◇ OPERATIONS: parameters + return values
- ◇ Substitutions: assignment, multiple assignment, parallel execution, IF-THEN-ELSE, CASE statement, PRE, SELECT, ANY

◇ Verification ◇

When a machine is presented, there are particular claims that are (implicitly) made about it:

- ◇ that it makes sense and is coherent
- ◇ that the initialisation establishes the invariant
- ◇ that operations preserve the invariant

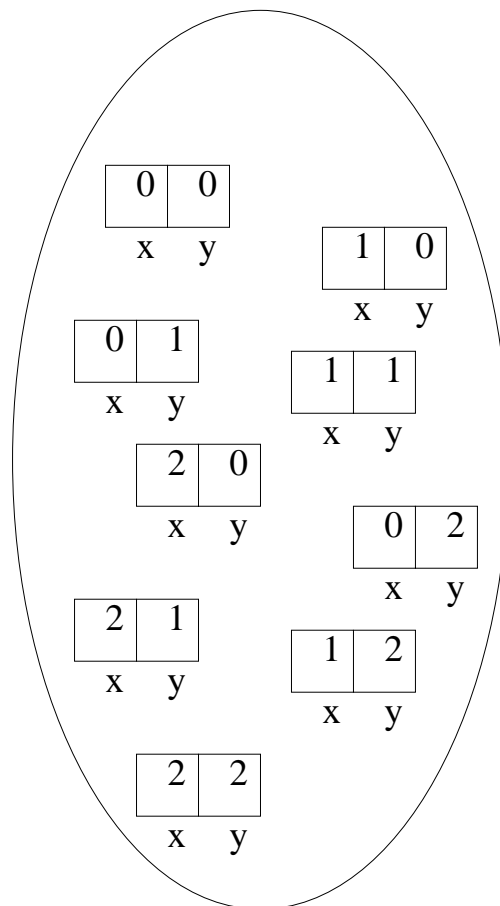
So we need to be able to make precise claims about AMN code.

◇ State Spaces ◇

The *state space* of a machine is the set of all possible states the machine can be in.

Example:

$x : 0..2$ & $y : 0..2$



◇ Changing State ◇

In principle, for a particular fragment of code we can say how each state should be transformed. This would be a specification.

A *statement* gives a way of transforming an initial into a final state.

A statement may be thought of as a *function* between initial and final states. (total, or partial?)

More generally, a statement may be thought of as a *relation* between initial and final states.

If there are infinitely many states then we have to encapsulate the way we want states to change in a general formula.

Exercise: Draw the function

$$f : (0..2 \times 0..2) \rightarrow (0..2 \times 0..2)$$

corresponding to the assignment

$$y := \max \{ y-x, 0 \}$$

◇ Simple substitution ◇

$x := E$

It executes by evaluating the expression E in the initial state, and then assigning the resulting value to variable x .

Examples:

- ◇ $y := 0$ transforms $(x=1, y=1)$ to $(x=1, y=0)$
- ◇ $y := y+1$ transforms $(x=1, y=1)$ to $(x=1, y=2)$
- ◇ $s := \{ \}$ transforms $s = \{ 4, 7 \}$ to $s = \{ \}$
- ◇ $s := s \setminus \{ 3, 9 \}$ transforms $s = \{ 4, 7 \}$ to $s = \{ 3, 4, 7, 9 \}$

◇ Predicate transformers ◇

We are generally interested in whether we have reached an ‘acceptable’ state rather than any particular state.

Rather than think in terms of single states, we think in terms of *sets of states*. These are described by *predicates*, which are assertions, or claims, about the state of the machine.

Examples of predicates include

$x > 1$, $x < y$, $s \neq \{ \}$.

The predicates are (implicitly) over all of the state variables.

Thinking in terms of predicates:

‘If the machine begins in a state in which Q is true, then it is guaranteed to finish in a state in which P is true’.

If $x := x+1$ is executed from a state in which $x < 5$ holds, then it is guaranteed to finish in a state in which $x < 6$.

◇ Exercise ◇

From what initial states is

$$y := \max \{ y-x, 0 \}$$

guaranteed to establish $y > 0$ in the final state?

◇ The notation $[S]P$ ◇

$[S]P$ is a predicate which holds of a state s if and only if every execution of S from s terminates in a state in which P holds.

- ◇ *Every* execution must reach P
- ◇ Termination must be guaranteed

To *guarantee* that P will hold in the final state, $[S]P$ must hold in the initial state.

Example:

$$[x:=x+1](x<6) = (x<5)$$

$[S]P$ is called the *weakest precondition* for S to establish P .

P is referred to as the *postcondition*.

◇ Weakest precondition ◇

If $Q = [S]P$ then if S is executed from a state in which Q is true then P is guaranteed to be true of any final state. (For predicates, equality is \Leftrightarrow).

If $Q \Rightarrow [S]P$ then if S is executed from a state in which Q is true then P is guaranteed to be true of any final state.

Example: $(x < 2) \Rightarrow [x := x + 1] (x < 6)$

This states that if $x := x + 1$ is executed in a state in which $x < 2$, then in the resulting state $x < 6$ will be true.

If $[S]P = \text{true}$, then S is always guaranteed to establish P , from any state.

Example:

$[\text{houseset} := \{\}] (\text{houseset} < : 1 .. \text{lasthouse})$

◇ Simple substitution ◇

$$\boxed{[x := E]P = P[E/x]}$$

If P is to hold of the final state, then $P[E/x]$ must hold of the initial state.

The assignment $x:=E$ is to be performed. If predicate P is to hold afterwards (when x has the value given by the value of E in the initial state) then P with E substituted for x should hold before.

Substitution: $P[E/x]$ is the predicate P with free occurrences of x replaced by the expression E .

- ◇ P and $P[E/x]$ are both predicates on the state variables.
- ◇ P is evaluated on the final values of the variables.
- ◇ $P[E/x]$ is evaluated on the initial values of the variables.

◇ Substitutions ◇

Substitution notation:

$P[E/x]$ replaces all free occurrences of x in P by E .

Examples:

$$\diamond (x = 5) [y/x] = (y=5)$$

$$\diamond (x > z) [y+3/x] = (y+3 > z)$$

$$\begin{aligned} \diamond (s : A \multimap B) [\{\}/s] \\ = \{ \} : A \multimap B \quad (= \text{true}) \end{aligned}$$

Thus

$$\diamond [s := s \ \backslash / \ \{x \mapsto y\}] (s : A \multimap B)$$

$$= (s \ \backslash / \ \{x \mapsto y\}) : A \multimap B$$

$$= s : A \multimap B$$

$$\& x:A \ \& y:B$$

$$\& (x / : \text{dom}(s) \ \text{or} \ x \mapsto y : s)$$

◇ Exercise ◇

4 Calculate the following preconditions:

$$\diamond [y := \max \{ y-x, 0 \}] (y > 0)$$

$$\diamond [x := y+1] (x > 4)$$

$$\diamond [x := y+1] (y > 4)$$

$$\diamond [x := y+1] (y > x)$$

$$\diamond [\text{houseset} := \text{houseset} \setminus \{\text{new}\}] \\ (\text{card}(\text{houseset}) < 17)$$

$$\diamond [\text{houseset} := \{\}] (\text{card}(\text{houseset}) < 17)$$

$$\diamond [\text{hh} := \min(\text{houseset})] (!\text{hh} . (\text{hh} < 163))$$

◇ Multiple assignment ◇

$$\boxed{[x, y := E, F]P = P[E, F/x, y]}$$

The notation $P[E, F/x, y]$ denotes the *simultaneous* substitution of E for x and F for y : all occurrences of x in P are replaced by E , and all occurrences of y in P are replaced by F .

For the assignment $x, y := E, F$ to achieve P in the final state, the predicate $P[E, F/x, y]$ should be true in the initial state.

Parallel assignments should be rewritten to multiple assignments before their preconditions can be calculated.

◇ Examples ◇

$$\begin{aligned} \diamond [x, y := 0, 0] (x \geq y) \\ &= (0 \geq 0) \\ &= \text{true} \end{aligned}$$

$$\diamond [x, y := 0, x+y] (y < 3) = (x+y < 3)$$

$$\begin{aligned} \diamond [x, y := 0, x+y] (x+y = 3) \\ &= (0+(x+y) = 3) \end{aligned}$$

$$\begin{aligned} \diamond [size, mx := size+1, \max(mx, new)] (mx \geq size) \\ &= \max(mx, new) \geq size+1 \end{aligned}$$

$$\begin{aligned} \text{Similarly, } [x_1, \dots, x_n := E_1, \dots, E_n] P \\ &= P[E_1, \dots, E_n / x_1, \dots, x_n] \end{aligned}$$

◇ Array substitution ◇

Arrays are functions...

and functions are just abstract variable types.

VARIABLES aa

INVARIANT $aa \in 1..10 \mapsto \mathbb{N}$

declares a 10 element array of natural numbers.
It does not even need to be total

$aa(i) := E$

This substitution evaluates the expression E in the initial state, and replaces the maplet $i \mapsto aa(i)$ in aa with $i \mapsto E$.

It is shorthand for the simple assignment to the variable aa :

$aa := aa \leftarrow \{ i \mapsto E \}$

Its effect on predicates is therefore given by

$$[aa(i) := E]P = P[aa \leftarrow \{ i \mapsto E \} / aa]$$

◇ Example ◇

$$\begin{aligned} & [aa(4) := 3](aa(4) < 8) \\ &= (aa(4) < 8)[aa \triangleleft \{4 \mapsto 3\} / aa] \\ &= (aa \triangleleft \{4 \mapsto 3\})(4) < 8 \\ &= 3 < 8 \\ &= \text{true} \end{aligned}$$

$$\begin{aligned} & [aa(4) := 3](aa(4) < aa(2)) \\ &= (aa(4) < aa(2))[aa \triangleleft \{4 \mapsto 3\} / aa] \\ &= (aa \triangleleft \{4 \mapsto 3\})(4) < (aa \triangleleft \{4 \mapsto 3\})(2) \\ &= 3 < aa(2) \end{aligned}$$

But beware of simply substituting E for $aa(i)$ in the predicate itself:

$$\begin{aligned} & [aa(i) := 3](aa(i) < aa(2)) \\ &= (aa(i) < aa(2))[aa \triangleleft \{i \mapsto 3\} / aa] \\ &= (aa \oplus \{i \mapsto 3\})(i) < (aa \triangleleft \{i \mapsto 3\})(2) \\ &= 3 < ??? \end{aligned}$$

◇ Exercises ◇

What is

$$[aa(4) := 7](aa(i) < 5)$$

Harder:

What is

$$[aa(4) := 7](\sum_{ii=1}^4 aa(ii) = 26)$$

Even harder:

What is

$$[aa(aa(i)) := aa(i)](aa(i) = i)$$

◇ Example ◇

If we do an assignment $a(4) := 7$, what conditions are required to ensure that the resulting function a is partial and injective?

$$\begin{aligned} & [a(4) := 7] (a : \text{NAT}1 \multimap \text{NAT}) \\ &= (a \ll \{4 \mid \rightarrow 7\}) : \text{NAT}1 \multimap \text{NAT} \\ &= \{4\} \ll a : \text{NAT}1 \multimap \text{NAT} \\ &\quad \& 7 / : \text{ran}(\{4\} \ll a) \end{aligned}$$

◇ Verifying abstract machines ◇

When an abstract machine specification is produced, there are particular claims that are (implicitly) made about it:

- ◇ that it makes sense and is coherent;
- ◇ that there are some parameters it can be given (otherwise it is useless);
- ◇ that its deferred sets and constants can be instantiated;
- ◇ that there are states that meet the invariant (otherwise it cannot be implemented);
- ◇ that the initialisation establishes the invariant;
- ◇ that operations preserve the invariant;

The B-Method allows these claims to be expressed formally as logical proof obligations on the machine.

◇ Initialisation ◇

A MACHINE contains an invariant.

It is necessary to check that the machine guarantees that this invariant will always be true.

The first step is to prove that the initialisation T establishes the invariant: when the machine is switched on, its initial state satisfies the invariant.

Question:

If the initialisation substitution is T , and the invariant is I , what has to be proven?

Exercise: Prove this for the initialisation of PaperRound.

◇ Operation Consistency ◇

In order to ensure that the AMN specification of an operation

```
PRE P THEN S END
```

is consistent within the machine, it is necessary to prove that it preserves the invariant when it is invoked under precondition P.

The following has to be proved:

$$I \ \& \ P \Rightarrow [S] I$$

(this is where we have to know how to calculate $[S] I$).

If the machine is in a state in which the invariant holds and the precondition also holds, then it should also be in a state in which execution of S is guaranteed to achieve I: after executing S, the invariant must still be true.

◇ Paper_Round Example ◇

MACHINE Paperround

SETS HOUSE

VARIABLES papers, magazines

INVARIANT

papers<:HOUSE & magazines<:papers

INITIALISATION

papers := {} || magazines := {}

OPERATIONS

addpaper(pp) = PRE pp:HOUSE & pp/:papers
THEN papers := papers ∖ {pp} END;

addmagazine(m) = PRE m:papers THEN
magazines := magazines ∖ {m} END;

remove(hh) = PRE hh:HOUSE THEN papers
:= papers - {hh} || magazines := magazines
- {hh} END

END

◇ Example ◇

Example: the operation `addpaper` of the `PaperRound` machine:

$$\begin{aligned} & PaperRoundInv \wedge pp \in HOUSE \wedge pp \notin papers \\ & \Rightarrow [papers := papers \cup \{pp\}] PaperRoundInv \end{aligned}$$

To prove this, we begin by reducing the right hand side, to remove the AMN and obtain a logical assertion. (In this example we omit the type information)

$$\begin{aligned} & [papers := papers \cup \{pp\}](magazines \subseteq papers) \\ & = magazines \subseteq papers \cup \{pp\} \\ & \Leftarrow magazines \subseteq papers \\ & \Leftarrow PaperRoundInv \wedge pp \in HOUSE \\ & \quad \wedge pp \notin papers \end{aligned}$$

Exercise: Verify `remove`

◇ Separating proof obligations ◇

An invariant often consists of a number of clauses

$$I = (I_1 \ \& \ I_2 \ \& \ \dots \ \& \ I_n)$$

To prove

$$[S] (I_1 \ \& \ I_2 \ \& \ \dots \ \& \ I_n)$$

it is sufficient to prove each clause separately is established by S:

$$[S] I_1 \ \& \ [S] I_2 \ \& \ \dots \ \& \ [S] I_n$$

This is what the B-Toolkit does. It generates a number of proof obligations

$$I \ \& \ P \Rightarrow [S] I_j$$

for each operation, corresponding to the individual clauses of the invariant.

◇ Operations with output ◇

Operations may provide output to a number of parameters. These are listed within the operation definition as follows:

```
mm <-- opn(nn)
```

Type information does not need to be explicitly provided for output variables where it is implicit from the body of the operation.

The invariant of the machine is not concerned with the value of the output. Any output will be consistent with the invariant of the machine. Hence there are no proof obligations concerning the outputs of the operation.

◇ Query operations ◇

If an operation does not change the state of the machine, but only provides output, then its proof obligations will automatically be true.

The invariant I can refer only to the state variables.

If S does not update any of the state variables, then it will leave I unchanged.

Hence $[S]I = I$: for I to be true after the execution of S , it must be true before.

So $I \ \& \ P \Rightarrow [S]I$ follows automatically.

Since this is a trivial proof obligation, the B-Toolkit doesn't even generate it.

(see e.g. the proof obligations generated for paperquery — there aren't any)

◇ Parameterised machines ◇

One way of building generic machines is to provide machines with *parameters*. The parameters for a machine are listed with the name of the machine.

MACHINE

PaperRound (maxround)

Parameters may be either set valued or scalar valued.

If a machine has parameters, then the logical properties of these parameters are specified in a CONSTRAINTS clause.

CONSTRAINTS

$$\textit{maxround} \in \mathbb{N}_1 \wedge \textit{maxround} > 20$$

These are properties concerning the size of such sets, or membership of scalar parameters in set parameters.

◇ Parameters ◇

There is a proof obligation associated with a constraints clause: that there are values which can be provided for the parameters which meet the constraints.

Proof obligations: if the body of the CONSTRAINTS clause is C , and the parameters are $x_1 \dots x_n$ then the following must be established:

$$\boxed{\exists x_1, \dots, x_n \bullet C}$$

In other words, there exist some values for x_1 up to x_n for which the constraint C holds.

PaperRound example:

$$\exists \mathit{maxround} \bullet \\ (\mathit{maxround} : \mathit{NAT1} \& \mathit{maxround} > 20)$$

...and this is true.

◇ SETS and CONSTANTS ◇

The other way of introducing sets into a machine definition is by means of the SETS clause.

SETS *HOUSE; RESPONSE = {yes, no}*

Other items which can be referred to in a read-only fashion are introduced in the **CONSTANTS** clause. These can denote single values, and can also denote function types.

CONSTANTS *lasthouse*

The **PROPERTIES** clause contains information about the logical and defining properties of the sets and constants given in the SETS and CONSTANTS clauses.

PROPERTIES

card(HOUSE) > maxround
∧ lasthouse ∈ HOUSE

◇ More proof obligations ◇

There are two more proof obligations. In a machine with

SETS St

CONSTANTS k

PROPERTIES B

- ◇ Firstly, in order for the machine to have any valid values for its parameters at all, it must always be possible, whatever parameters are provided to the machine, to find appropriate sets and constants:

$$C \Rightarrow (\exists St, k \bullet B)$$

(B4Free: VALUES clause in implementations)

- ◇ Secondly, when all the parameters have been set, there must be some valid state of the machine—i.e. a state that meets the machine invariant I :

$$B \wedge C \Rightarrow \exists v \bullet I$$

Exercise: do these for PaperRound. Are they true?

◇ Summary ◇

All the clauses that are listed in a machine are listed here.

MACHINE N(p)

CONSTRAINTS C

SETS St

CONSTANTS k

PROPERTIES B

VARIABLES v

INVARIANT I

INITIALISATION T

OPERATIONS

 y \leftarrow op(x) =

 PRE P

 THEN S

 END;

 ...

END

◇ Proof Obligations ◇

The following must be proven for a machine to show that it makes sense:

1. $\exists p \bullet C$
2. $C \Rightarrow (\exists St, k \bullet B)$
3. $B \wedge C \Rightarrow \exists v \bullet I$
4. $B \wedge C \Rightarrow [T]I$
5. $B \wedge C \wedge I \wedge P \Rightarrow [S]I$ for each operation **PRE P THEN S END**

1–3 are concerned with consistency of the context of the machine. 4 & 5 concern consistency between the static and dynamic specifications.

B4Free generates proof obligations of any machine automatically. Those that cannot be proven may indicate an inconsistent operation.

In B4Free you have to prove 1,2,3 only for implementations. ProB can find counter examples to 4,5 and will tell you if it cannot find values satisfying 1,2,3.

◇ Further weakest preconditions ◇

Given an AMN statement S and a postcondition P , you want ways of working out what the weakest precondition $[S]P$ required for S to guarantee P .

We will be considering:

- ◇ conditional
- ◇ preconditioned substitution
- ◇ choice
- ◇ unbounded choice

◇ IF THEN ELSE ◇

```
IF
  P
THEN
  S1
ELSE
  S2
END
```

From which initial states does

```
IF  $x < 7$ 
THEN  $x := x + 5$ 
ELSE  $x := x + 1$ 
END
```

guarantee that $x > 10$ in the final state?

Weakest precondition:

What is $[IF P THEN S1 ELSE S2 END]Q$

◇ IF THEN ELSE ◇

[IF P THEN S1 ELSE S2 END]Q

=

$P \wedge [S1] Q \vee \neg P \wedge [S2] Q$

=

$P \Rightarrow [S1] Q \wedge \neg P \Rightarrow [S2] Q$

What about:

[IF P THEN S1 END]Q

=?

◇ Exercise ◇

Prove that the following new operation of PaperRound preserves the invariant:

$$\text{magazines} \subseteq \text{papers} \wedge \text{lasthouse} \notin \text{magazines}$$

PRE $hh : \text{HOUSE}$

THEN

IF

$$hh = \text{lasthouse}$$

THEN

$$\text{papers} := \text{papers} - \{hh\}$$

ELSE

$$\text{magazines} := \text{magazines} - \{hh\}$$

END

END

◇ Preconditioned Substitution ◇

The assignment `PRE P THEN S END` executes `S` if the precondition `P` is true, otherwise anything can happen (including non-termination — not reaching a final state at all).

From which initial states does
`PRE x < 7 THEN x := x+5 END`
guarantee that `x > 10` in the final state?

Weakest precondition:

What is $[\text{PRE } P \text{ THEN } S \text{ END}]Q$

◇ Choice substitution ◇

```
CHOICE
  S1
OR
  S2
END
```

This statement executes by executing either S1 or S2. The user of the computer has no control over which is executed.

For which initial states does
CHOICE $x := x + 1$ OR $x := x - 1$ END
guarantee the final state satisfies $x > 7$?

Weakest precondition:

What is $[\text{CHOICE } S1 \text{ OR } S2 \text{ END}]P$

◇ **Exercise** ◇

What does the following substitution do? **CHOICE**

$$\text{magazines} := \text{magazines} \cup \{hh\}$$

OR

$$\text{papers} := \text{papers} - \{hh\}$$

END

Calculate the precondition required to guarantee each of the following:

◇ $\text{magazines} \subseteq \text{papers}$

◇ $\text{houseset} \not\subseteq \text{magazines}$

◇ Unbounded Choice ◇

```
ANY      xx
WHERE    P
THEN     S
END
```

This statement chooses an arbitrary xx which satisfies P , and then executes the statement S

For which initial states does

```
ANY xx
WHERE xx > yy
THEN zz := xx*xx
END
```

guarantee that the final state has $zz > 12$?

What is $[ANY\ xx\ WHERE\ P\ THEN\ S\ END]Q$?

◇ Exercise ◇

$[ANY\ xx\ WHERE\ P\ THEN\ S\ END]Q$

$= \forall xx.(P \Rightarrow [S]Q)$

Exercise: Does the operation

PRE *lasthouse* \notin *papers*

THEN

ANY *hh*

WHERE *hh* \in *papers*

THEN *magazines* := *magazines* \cup {*hh*}

END

END

preserve the invariant

magazines \subseteq *papers*

\wedge *lasthouse* \notin *magazines*

Do a proof if it does or give an example if it doesn't.

◇ Sequences (7.8) ◇

- ◇ Notation: $[e_1, \dots, e_k]$
- ◇ Represented as a special function:
 $\{ 1 \mapsto e_1, \dots, k \mapsto e_k \}$
- ◇ Empty Sequence: $[]$
- ◇ $\text{seq}(S)$ set of sequences with elements from S
- ◇ $\text{seq1}(S)$ set of non-empty sequences with elements from S

- ◇ How to rewrite $\text{seq}(S)$ using “classical” notation?

◇ Operators for Sequences ◇

- ◇ `size(s)` get size of a sequence
- ◇ `rev(s)` reverse a sequence
- ◇ `first(s)` get first element
- ◇ `last(s)` get first element
- ◇ `s ^ t` concatenate two sequences
- ◇ `conc(ss)` concatenate all sequences in `ss`
- ◇ `E->s` prepend an element `E` to `s`
- ◇ `s<-s` append an element `E` to `s`
- ◇ `front(s)` front of sequence
(all but last element)
- ◇ `tail(s)` tail of sequence
(all but first element)
- ◇ `s/|\n` take first `n` elements of sequence `s`
- ◇ `s\|/n` drop first `n` elements from `s`
- ◇ Exercise: rewrite above using “classical” notation?

◇ Results Example ◇

```
MACHINE                               Results
SETS                                  RUNNER
VARIABLES                             finish
INVARIANT                             finish : iseq(RUNNER)
INITIALISATION                        finish := <>
OPERATIONS
finished(rr) =
  PRE rr : RUNNER & rr /: ran(finish)
  THEN finish := finish <- rr
  END;
rr <-- query(pp) =
  PRE pp : NAT1 & pp <= size(finish)
  THEN rr := finish(pp)
  END;
disqualify(pp) =
  PRE pp : NAT1 & pp <= size(finish)
  THEN finish :=
    (finish /|\ pp-1) ^ (finish \|\ pp)
  END;
ss <-- medals = ss := finish /|\ 3
END
```

◇ MoreSequences ◇

- ◇ $\text{seq}(S)$ set of sequences with elements from S
- ◇ $\text{seq1}(S)$ set of non-empty sequences with elements from S
- ◇ $\text{iseq}(S)$ set of injective sequences with elements from S
- ◇ $\text{iseq1}(S)$ set of non-empty injective sequences with elements from S
- ◇ $\text{perm}(S)$ set of bijective sequences with elements from S (permutations)

◇ Using ProB ◇

- ◇ Size of SETS
 - Enumeration $S = \{a, b, c\}$
 - Set Default Preference
 - `scope_S == 1..3` in DEFINITIONS
- ◇ NAT versus NATURAL
- ◇ MININT, MAXINT
- ◇ SudokuSETS

◇ Using B4Free ◇

- ◇ Selected Hypotheses
 - too small: goal cannot be proven
 - too big: prover gets lost
- ◇ Assume Hypothesis (ah)
- ◇ Add to Properties (be careful !!)
- ◇ Rewrite (eh, he)
- ◇ Instantiate forall (ph)

(See separate set of slides on B4Free)

◇ Software system development ◇

Abstract machines are *specifications* of systems.

In order to reach code, it is necessary to provide *refinements* for them.

Refinements are a step towards saying *how* the state variables are to be implemented, and *how* the operations are to be carried out.

Example:

$\text{ans} := \text{SIGMA}(\text{zz}).(\text{zz} : \text{numset} \mid \text{zz})$

is part of a *specification*.

This might be *implemented* by maintaining a running total:

$\text{ans} := \text{runtot}$

Of course, we have to be confident that *runtot* is maintaining the right information!

◇ Sequential composition ◇

Refinement machines are allowed to contain sequencing.

$S ; T$ performs S and then T .

Q: For which initial states does
 $x := x + 1 ; y := y - x$
guarantee that the final state satisfies $x > y$?

Q: For which initial states does
(CHOICE $x := x + 1$ OR $y := y - x$ END) ;
 $x, y := y, x$
guarantee that the final state satisfies $x > y$?

What is $[S ; T]P$?

Exercise: calculate

$[x := x - y ; y := y + x ; x := y - x] (x = A \ \& \ y = B)$

◇ Local variables ◇

Local variables may be introduced into an AMN statement by means of a VAR statement.

(Local variables are useful once you have intermediate states.)

```
VAR vv IN S END
```

Example:

```
SWAP = VAR tt
        IN
        tt := xx;
        xx := yy;
        yy := tt;
        END
```

What is $[\text{VAR } vv \text{ IN } S \text{ END}]P$?

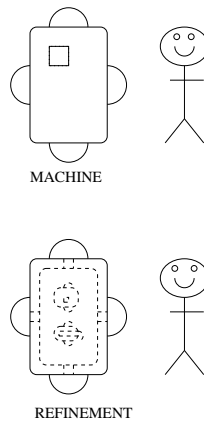
exercise: calculate

$[\text{SWAP}] (xx = A \ \& \ yy = B)$

◇ Refinements ◇

Refinements are provided for MACHINES.

A user should *not* be able to detect that an abstract machine has been replaced by an refinement.



A refinement must have the same interface as the machine it implements. It offers the same operations, with the same inputs and outputs.

The user cannot directly see the state inside the machine or the refinement. The user can interact with the machine only through its operations.

◇ Refinement machines ◇

REFINEMENT MiniMilkR

REFINES MiniMilk

With regard to this refinement relation, MiniMilkR is the concrete machine, and MiniMilk is the abstract machine.

Refinement machines are a step along the way to arriving at an implementation.

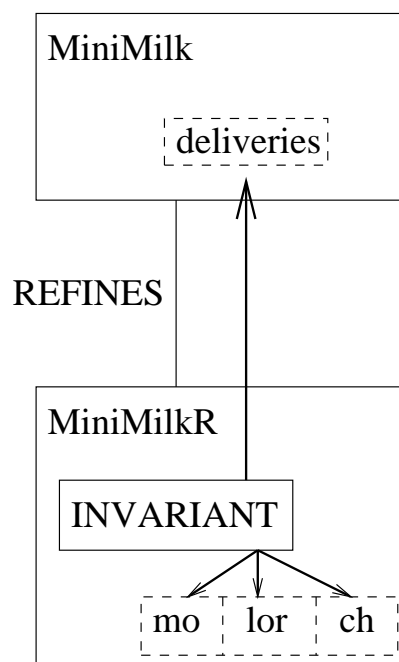
Refinement machines can contain all constructs allowed in abstract machines (abstract variable types, nondeterminism, etc), as well as sequential composition.

The state variables of a refinement must be related in the linking invariant to the variables of the machine it refines. This is similar to a REP invariant.

◇ Linking invariants ◇

The invariant of the refinement machine *links* the variables of the refined machine with the variables of the imported machines. This relation is called the *linking invariant*.

It states how the concrete state *corresponds to or implements* the abstract state.



◇ Refining operations ◇

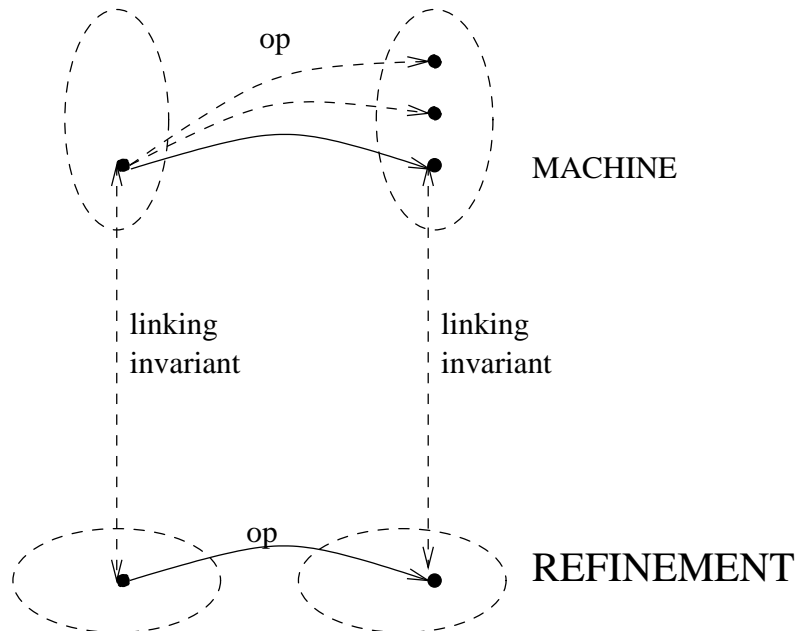
Each operation must provide exactly the same interface at both the abstract and the concrete level. Not only the operation names, but also the formal parameters must be identical.

Definitions of concrete operations must be given entirely in terms of the concrete state variables.

Every invocation of a concrete operation from a valid state must be matched by some invocation of the abstract operation, which preserves the linking invariant.

◇ Operations ◇

An implementation is *correct* if its operations preserve the linking invariant.



Also, the linking invariant must be initially true.

◇ Operations ◇

Each operation must provide exactly the same interface at both the abstract and the concrete level.

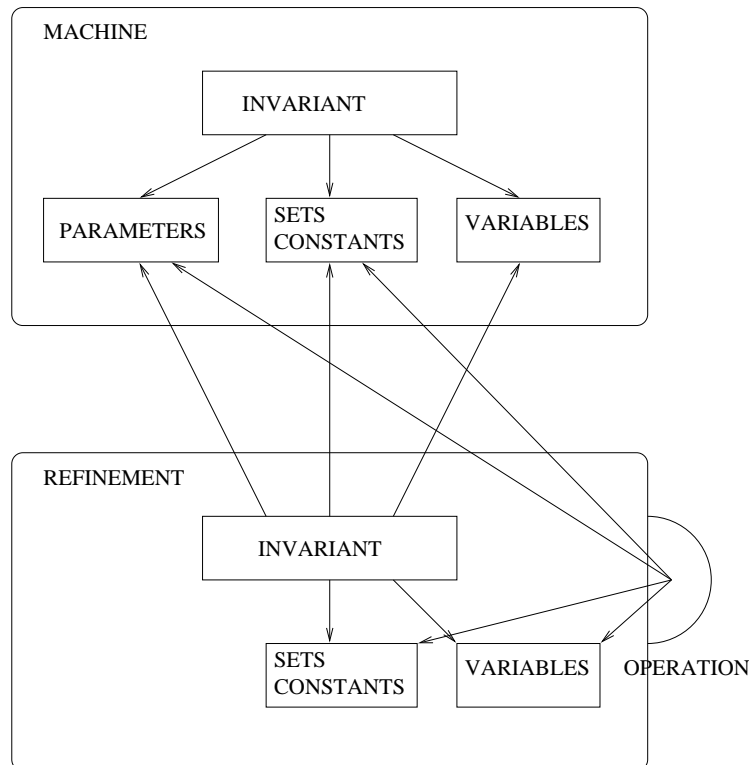
If the implemented operation provides output, then the machine operation should also be able to provide the same output.

Example: `lengthofround`

◇ Relating machines ◇

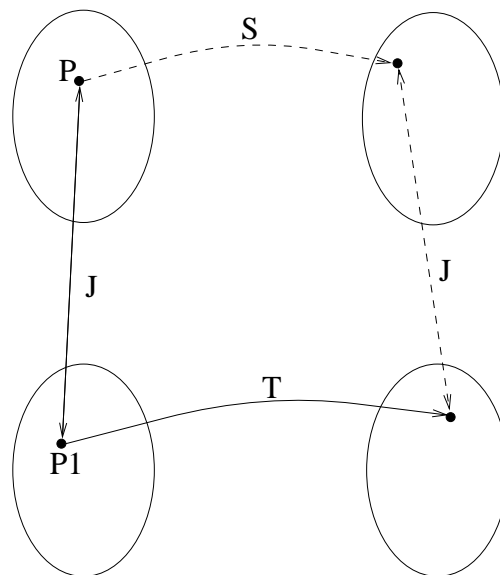
A refinement inherits and has access to the parameters, sets and constants of the machine it refines. These can all appear in its invariant and operations.

The invariant can also refer to the variables of the refined machine.



◇ Refining operations ◇

If the abstract operation is enabled, then so too should the concrete one be. Then the linking invariant must be preserved.



The refined operation only has to behave correctly when the abstract operation is enabled. (see `maxorder`.)

◇ Towards a proof obligation ◇

If T is the refined body of an operation, and S is the abstract body, then we require that whenever T does a state transformation, then it is possible for S to do a state transformation from a linked state, and end up in a linked state.

This is captured as follows:

$$J \Rightarrow [T] \text{not } [S] \text{not } (J)$$

Breaking down the formula:

- ◇ $[S] \text{not } (J)$ is true if S is guaranteed to establish $\text{not } (J)$
- ◇ $\text{not } [S] \text{not } (J)$ is true if some execution of S can establish J
- ◇ $[T] (\text{not } [S] \text{not } (J))$ is true if any execution of T reaches a state where some execution of S can establish J .

Thus any execution of T can be matched by S

◇ Example ◇

A fragment of the refinement:

The *ch* variable, in the `addtoround` operation:

J : `ch : dom(deliveries)`

S : `deliveries(hh) := nn`

T : `ch := hh`

[S] (`not(J)`) :

(`not [S] (not(J))`) :

[T] (`not [S] (not(J))`) :

J => [T] (`not [S] (not(J))`) :

◇ Preconditions ◇

opn = PRE P THEN S END

[T]not([S](not(J))) need be true only for legal states (i.e. where the invariants are true), and the abstract precondition is true

- ◇ If the abstract precondition is false, then T may do anything (and the result will still be a refinement).
- ◇ If the abstract precondition is true, then the concrete one should also be.

If I and J are the invariants of the abstract and concrete machines respectively, then the full rule (apart from constraints and properties) for PRE P1 THEN T END to refine PRE P THEN S END is:

- ◇ $(I \ \& \ J \ \& \ P) \Rightarrow P1$
- ◇ $(I \ \& \ J \ \& \ P) \Rightarrow [T]not([S](not(J)))$

Example: updating lor in the addtoround operation.

◇ Dealing with output ◇

Outputs are updated by the body of an operation. The same output variable is part of the description of both the abstract and the concrete machine (since they must have the same interface).

The abstract and concrete copies of output variables must be considered separately. This is achieved by renaming the concrete outputs y to y' , so the concrete operation body T becomes $T[y'/y]$. The requirement is that abstract and concrete outputs should be the same after the operation: so $y = y'$ is added to the linking invariant:

$$\diamond (I \ \& \ J \ \& \ P) \Rightarrow P1$$

$$\diamond (I \ \& \ J \ \& \ P) \Rightarrow \\ [(T[y'/y])] \text{not} [S] \text{not} (J \ \& \ y=y')$$

◇ Output example ◇

```
nn <-- maxorder =  
  PRE  
    deliveries /= { }  
  THEN  
    nn := max(ran(deliveries))  
  END
```

is refined by

```
nn <-- maxorder =  
  PRE  
    true  
  THEN  
    nn := mo  
  END
```

We have to prove that this is a refinement.

◇ Refining nondeterminism ◇

One form of refinement is to remove or reduce nondeterminism. For example, if any element of a set can be chosen as output, then the implementor is allowed to determine how the choice is resolved.

```
hh <-- choose =  
  PRE deliveries /= { }  
  THEN hh :: dom(deliveries)  
  END
```

is refined by

```
hh <-- choose =  
  PRE true  
  THEN hh := ch  
  END
```

with

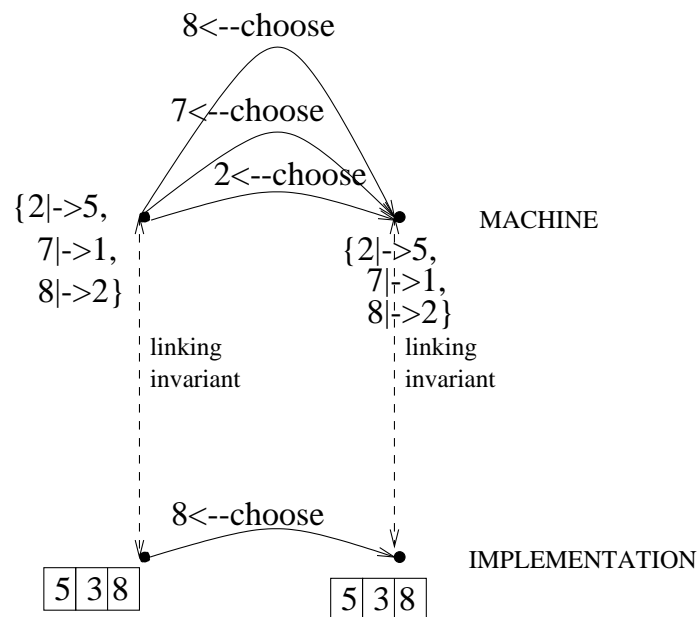
```
J : ch : dom(deliveries)
```

Exercise: How does the proof obligation work out in this case?

◇ Implementing Choose ◇

Whenever a particular output may be provided by the concrete operation, the abstract operation can provide the same output.

The specification states what is *allowed*, and the implementation states what will happen.



Question: what happens if choose is selected when deliveries is empty?

◇ Summary ◇

What's required for refinement:

- ◇ a linking invariant.
- ◇ whenever the concrete operation is invoked (possibly with some input):
 - any output it can provide must be possible for the abstract version of the operation
 - any state it can reach must be matched (via the linking invariant) by the abstract operation

This ensures that any sequence of interactions with the concrete implementation is also possible for the abstract machine. Hence a user can never know that the abstract machine has been replaced by the implementation.

◇ IMPLEMENTATIONs ◇

An IMPLEMENTATION machine is intended to be executed on a computer. It cannot contain arbitrary AMN, since some of this is not executable.

It therefore contains only a restricted number of AMN constructs. These correspond directly to elements that can be found in real programming languages executed on computers. This allows implementations to be turned into code.

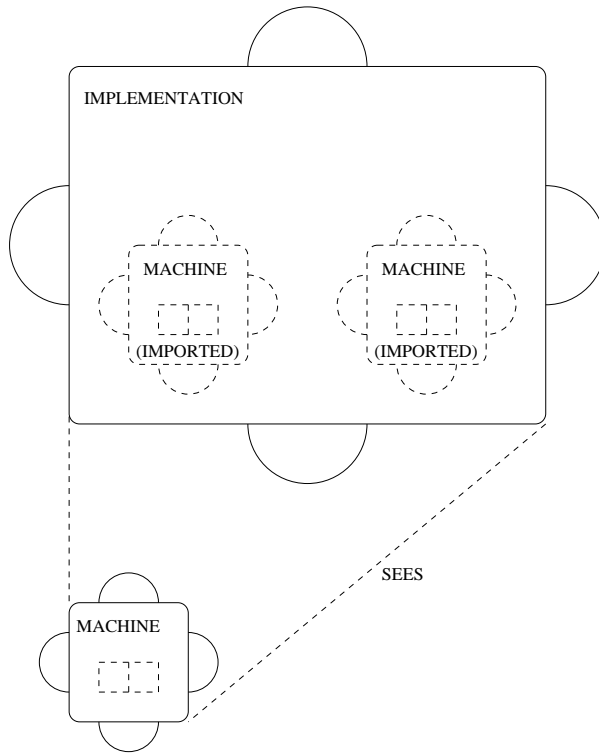
Implementations can also make use of constructs not permitted in specifications, notably sequential composition and loops. These constructs are concerned with *how* rather than *what*, and are not appropriate in specifications.

IMPLEMENTATIONs refine MACHINEs or REFINEMENTs.

◇ Incorporating other machines ◇

Implementations make use of other machines:

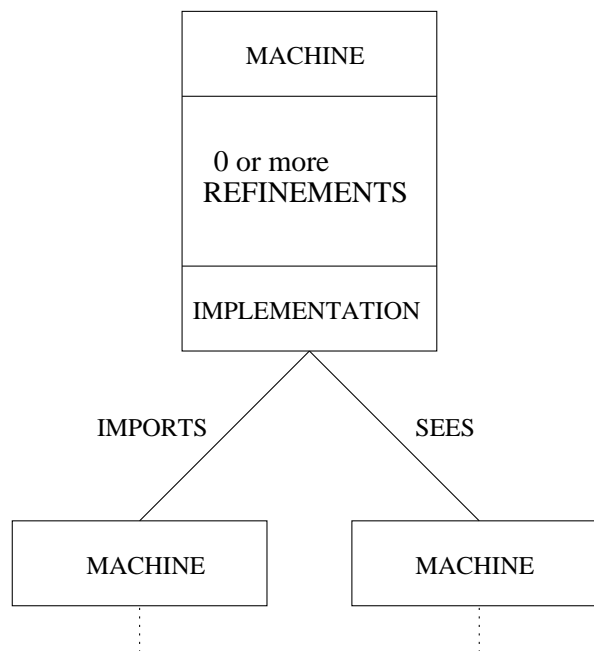
- ◇ IMPORTS: importing their own private copy of a machine which is completely under their control (cf the *part of* relationship);
- ◇ SEES: receiving information from another machine which is not completely under their control (cf *communicates with*)



◇ Structuring developments ◇

IMPORTS and SEES are crucial for **structuring** developments. [Structuring mechanisms for specifications will be covered later in the course.]

The machines that are imported and seen must *abstract machines*, i.e. specifications.



◇ Incorporated machines ◇

Any machines incorporated into an implementation can be accessed only via their operations.

IMPORTed machines have all of their operations available. Their state can be access and changed by means of the operations.

Question: why is *direct* access and updating of imported variables not allowed?

SEEn machines have available only their operations that do not change the state. This includes query operations, and functions provided by the SEEn machine.

Only abstract MACHINES (i.e. *specifications*) can be IMPORTed or SEEn.

When one machine is incorporated into another, the required information is *what* it does, not how it does it.

◇ Variables ◇

Implementations do not have their own state!

Any state that they require must be in imported machines.

This means that the VARIABLES clause may not appear in implementations.

An implementation machine does not have any variables of its own. Any state that it controls will be in the imported machines.

It can only interact with imported machines by means of their operations. It does not have direct access to their state variables.

◇ Implementing operations ◇

Operations in implementations *do not have preconditions*. (i.e. they are always permitted). They are required to implement the corresponding machine operation `PRE P THEN S END` only when `P` is true.

Example:

```
ans <-- div(xx, yy) =  
PRE yy:NAT1 & xx:NAT  
THEN ans := (xx*xx) / yy  
END
```

is implemented by

```
ans<--div(xx, yy) = ans:=(xx*xx)/yy
```

A user of the machine providing this operation is required to supply inputs in line with the precondition.

◇ Another implementation ◇

```
ans <-- div(xx, yy) =  
PRE yy:NAT1 & xx:NAT  
THEN ans := (xx * xx) / yy  
END
```

is also implemented by

```
ans <-- div(xx, yy) =  
IF yy = 0  
THEN ans := 17  
ELSE ans := (xx * xx) / yy  
END
```

◇ Operation definition ◇

Permissible constructs in implementation operation definitions:

- ◇ VAR *xx* IN *S* END (local variable)
- ◇ *v* := *E* where each *v* is either a local variable of the operation, or an output variable.
- ◇ *S*1 ; *S*2
- ◇ IF *E* THEN *S*1 ELSE *S*2 END
- ◇ WHILE *E* DO *S*
 VARIANT *v* INVARIANT *I* END
(to be covered later in the course)
- ◇ operations from imported machines
- ◇ query operations of seen machines

◇ Forbidden constructs ◇

Nondeterminism is resolved during implementation. Implementations must contain only constructs that can be supported on a computer.

This means that the following **cannot** appear in implementations:

- ◇ nondeterministic substitutions
(CHOICE, ANY, $xx :: E$)
- ◇ simultaneous substitution ($||$)
- ◇ preconditioned substitutions
- ◇ assignments involving abstract variable types

◇ Notes ◇

- ◇ the INVARIANT clause gives the relationship between the variables in the abstract machine which is being refined and the variables in the imported machine(s). This essentially explains why the refinement works;
- ◇ the INITIALISATION clause can initialise the imported machines, by means of their operations;
- ◇ all operations make use of operations of the imported machine.

◇ Handling state ◇

The way an implementation manages state variables is by means of IMPORTed machines.

Such machines must themselves be implemented.

There are special library machines that model the behaviour of a state variable. Library machines are specification machines.

They offer operations which enable the state they maintain to be updated in a variety of ways, and the current value in the state to be read.

Library machines are IMPORTed into implementations in the same way as other machines.

The B Toolkit already has built into it the ability to implement library machines directly in C code.

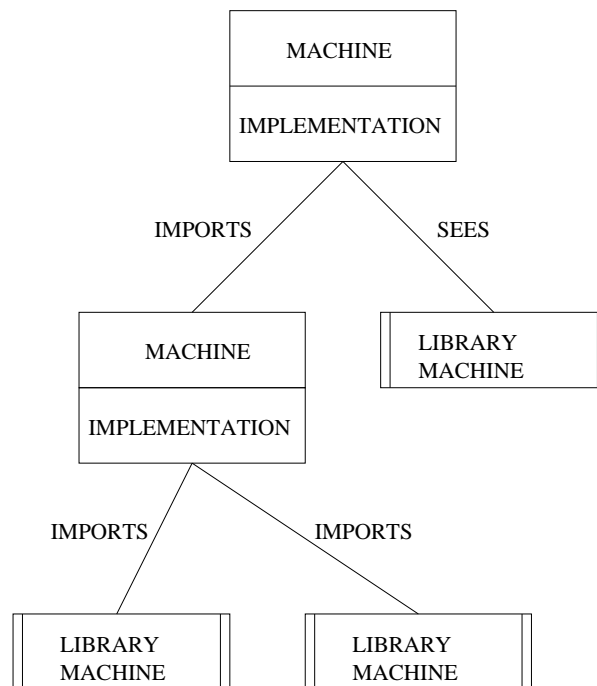
They are not refined further.

◇ Library machines ◇

The B-Toolkit provides a library of machines that implementations commonly require: support for state, and for types.

Developments do not need to provide implementations for them. The B-Toolkit generates code for them.

Development stops at library machines.



◇ A library machine: Nvar ◇

The machine Nvar is a generic library machine that provides the facility of a natural number state variable.

```
MACHINE Nvar(maxint)
```

```
VARIABLES Nvar
```

```
OPERATIONS:
```

```
◇ vv <-- VAL_NVAR =  
  vv := Nvar
```

```
◇ STO_NVAR(vv) =  
  PRE vv : 0..maxint  
  THEN Nvar := vv  
  END
```

There are many other operations for querying and updating the state:

```
◇ bb <-- PRE_INC_NVAR
```

```
◇ INC_NVAR
```

◇ Introducing library machines ◇

In order to introduce a particular instance of a generic library machine into a development, it is necessary to give it a name:

Variable machines are renamed.

e.g. `nn.Nvar(100000)` describes a particular instance of the `Nvar` machine with everything prefixed by `nn`, and which can handle numbers no greater than 100000.

As expected, the libraries provide *specifications* for these library machines. In other words, they are understood as MACHINES.

See `MiniMilkRI.imp`

◇ nn.Nvar ◇

When appearing in the IMPORTS clause, the generic machine should have its name prefixed, and its parameters provided.

```
IMPORTS nn.Nvar(100000)
```

The machine's variable is also prefixed with the name: nn.Nvar.

All operations are also prefixed with the name:

```
◇ vv <-- nn.VAL_NVAR =  
  vv := nn.Nvar
```

```
◇ nn.STO_NVAR(vv) =  
  PRE vv : 0..maxint  
  THEN Nvar := vv  
  END
```

◇ Nvar ◇

The Nvar machine also provides many other operations (see handout):

- ◇ arithmetic operations
- ◇ comparison operations
- ◇ queries on whether operation preconditions are true
- ◇ save/restore operations

These are pretty much all the operations you would hope for.

In fact, all operations (apart from save and restore) are simply provided for the sake of efficiency, and could be implemented using `STO_NVAR` and `VAL_NVAR`.

◇ Library machines ◇

There are many library machines provided by the B Toolkit. They are kept in SLIB, and may all be introduced into developments.

Library machines supporting variables are as follows:

- ◇ natural numbers `Nvar(maxint)`
- ◇ arrays of natural numbers `Narr(maxint,maxidx)`, maintaining a variable
`name_Narr:1..maxidx-->0..maxint`
- ◇ Partial functions `Nfnc(maxint,maxfld)`, maintaining
`name_Nfnc:1..maxfld+-->0..maxint`
- ◇ Sequences of natural numbers
`Nseq(maxint,maxsize)`, maintaining
`name_Nseq:seq(0..maxint)`
- ◇ scalar value `Vvar(VALUE)`, maintaining
`name_Vvar:VALUE`

◇ More library machines ◇

- ◇ arrays of scalar values `Varr(VALUE,maxidx)`, maintaining
`name_Varr:1..maxidx-->VALUE`
- ◇ strings, functions to scalar values, sequences of scalars, etc

Library machines provide all the operations you would expect to access and manipulate its state.

Also I/O machines and `file_dump` machines.

◇ AtelierB Base and Library Machines ◇

- ◇ Base Machine: no development in B; directly written in target code
 - BASIC_ARRAY_VAR: one dimensional array
 - BASIC_ARRAY_RANGE: two dimensional array
 - BASIC_IO: IO in vt100 style
- ◇ Library Machine: development in B; code generated as for user developments

◇ AtelierB Library Machines ◇

- ◇ L_ARITHMETIC1: one dimensional array
- ◇ L_ARRAY1: implements a B function using a one-dimensional array
- ◇ L_PNFC: implementation of a partial function
- ◇ L_SEQUENCE: implementation of a sequence
- ◇ L_SET: implementation of a set
- ◇ and many more ...

◇ SEES ◇

In an implementation, `M2 SEES M1` means that `M2` may invoke query operations of `M1`. However, `M2` may not change the state of `M1`, so update operations are not permitted.

A machine may appear in a number of `SEES` clauses of other machines.

In contrast, a machine may only be `IMPORTed` by one other machine.

Question: why?

Machines for types are generally accessed through `SEES`. This allows a number of machines to access the same type machine.

Providing support for types means providing a type and operations on it.

◇ Bool_TYPE ◇

The library machine Bool_TYPE provides support for handling booleans.

Library machines make use of this machine (via SEES Bool_TYPE), so machines which IMPORT such library machines will also need to SEE it if using the query operations.

```
MACHINE      Bool_TYPE
SETS        BOOL = { FALSE, TRUE }
```

with operations

OPERATIONS

```
bb <-- CNJ_BOOL(cc,dd) = ...
bb <-- DIS_BOOL(cc,dd) = ...
bb <-- NEG_BOOL(cc) = ...
vv <-- BTS_BOOL(bb) = ...
```

Type machines provide a type set (BOOL in this case) and a collection of operations for manipulating elements of that type. They have no internal state.

◇ Multiplication as a Loop ◇

Computing $RR * SS$:

```
res = 0;  
rr = RR;  
while (rr > 0) {  
    rr = rr - 1  
    res = res+SS }  

```

Is it correct? Why?

◇ Russian Multiplication ◇

Start off with your two numbers. Repeatedly halve one (rounding down) and double the other. Add up the doubled values that correspond to odd numbers of the halved numbers. The result is the product of the two numbers.

Example:

49	82
24	164
12	328
6	656
3	1312
1	2624
0	5248

$$82 + 1312 + 2624 = 4018$$

Why does it work?

◇ A code example ◇

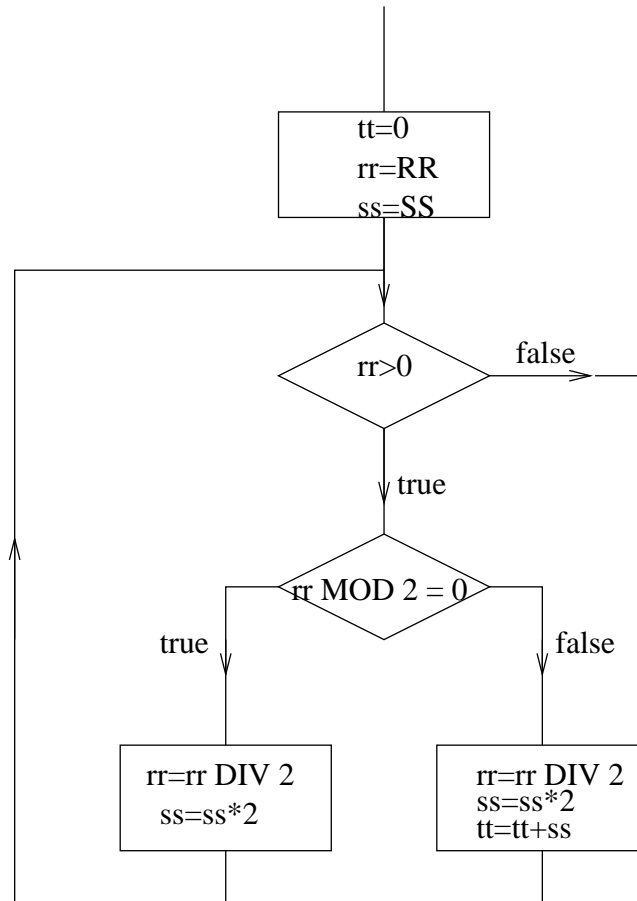
Here's a pseudocode program to do the same thing: multiply RR and SS:

```
tt = 0;
rr = RR;
ss = SS;
while (rr > 0) {
    if rr MOD 2 = 0
    then rr=rr/2 || ss = ss*2
    else rr=rr/2 || ss = ss*2 || tt=tt+ss }
```

Is it correct?

◇ Executing the loop ◇

A flow graph:



How might you reason about this loop?

Why do you think it's correct?

◇ Loops ◇

The general form in AMN :

WHILE E

DO S

END

- ◇ E is a *condition* or boolean expression on the state space. It is called the *guard* of the loop.
- ◇ If the loop is executed in a state s in which E is false, then the loop terminates immediately without changing the state.
- ◇ If the loop is executed in a state s in which E is true, then the loop executes S to reach a state s' from which the loop is again executed.
- ◇ In general it is possible that a loop never terminates execution: i.e. that the guard is always true.
- ◇ Execution of a loop passes through a sequence of intermediate states

◇ Multiplication Loop ◇

Specifying multiplication:

```
tt := RR * SS
```

This is what we want to be true when the program halts.

Is this an adequate specification of the program?

Under what conditions does the program meet this specification?

◇ Loop Invariants ◇

If a loop is executed from a state s in which I holds, and execution of the body of the loop S preserves I , then every intermediate state in the execution, and the final state, satisfies I . In this case I is a *loop invariant*

```
WHILE E
DO S
INVARIANT I
END
```

The body of the loop S is executed only if E is true. The claim that it preserves the invariant I is precisely:

$$\boxed{!l. (I \ \& \ E \Rightarrow [S] I)}$$

where l is the list of variables modified within the loop (i.e. on the left-hand side of a substitution).

In the example, a suitable invariant would be $tt + (rr * ss) = RR * SS$

◇ Final state ◇

If a loop terminates, then the guard E must be false. The invariant I remains true, as it is true in every state reached by the loop.

Hence any final state obtained must satisfy I & not E .

This should imply the desired postcondition P .

$$\boxed{!1. (I \ \& \ \text{not } E \Rightarrow P)}$$

In the example, the desired postcondition is that $tt = RR * SS$.

This will be true in any final state, since

$$\begin{array}{ll} tt + (rr * ss) = RR * SS & (I) \\ \& \text{not } (rr > 0) & (\text{not } E) \\ \Rightarrow & \\ tt = RR * SS & (P) \end{array}$$

◇ Termination ◇

In order to ensure that the loop will terminate, we must prove that it moves closer towards a final state on each iteration.

A *variant* is a value v associated with states which should be a natural number, and which strictly decreases every time the body of the loop is executed. The value of the variant is an upper bound on the number of iterations remaining for the loop. It is written in a `VARIANT` clause.

In general it can be any expression written in terms of the state variables.

◇ !1.(I & E => v : NAT)

◇ !1.(I & E & v = γ => [S] (v < γ))

Exercise: Prove these for variant `rr`.

◇ Multiplication1 ◇

Computing $RR * SS$:

```
tot := 0; x:= a; y:= b;
WHILE x>0 DO
  IF x mod 2 = 1 THEN
    tot := tot+y
  END;
  x := x/2;
  IF x>0 THEN y := y*2 END
INVARIANT x:NAT & y:NAT &
           tot:NAT & tot+x*y = a*b
VARIANT x
END;
res := tot
```

◇ Example Revisited ◇

The (slightly different) loop with its variant and invariant:

```
tt:=0; rr:=RR; ss:=SS;
WHILE
  rr>0
DO
  IF rr MOD 2 /= 0 THEN tt:=tt+ss END;
  rr:=rr/2;
  ss:=ss*2
INVARIANT
  tt+(rr*ss) = RR * SS & rr:NAT
VARIANT
  rr
END
```

The loop contains not only the code to be executed, but also the reasons why the loop is guaranteed to terminate and to be correct.

Observe that the clause `rr:NAT` is part of the invariant in order to guarantee termination.

◇ Loop Semantics ◇

If the following five conditions hold in the initial state, then the loop is guaranteed to terminate and establish P .

1. $\text{!}1.(I \ \& \ E \Rightarrow [S] I)$
2. $\text{!}1.(I \ \& \ \text{not } E \Rightarrow P)$
3. $\text{!}1.(I \ \& \ E \Rightarrow v : \text{NAT})$
4. $\text{!}1.(I \ \& \ E \ \& \ v = \gamma \Rightarrow [S] (v < \gamma))$
5. I

\Rightarrow

$$\left[\begin{array}{l} \text{WHILE } E \\ \text{DO } S \\ \text{INVARIANT } I \\ \text{VARIANT } v \\ \text{END} \end{array} \right] P$$

If the five conditions are established, then the loop is guaranteed to establish P

◇ Example: array initialisation ◇

```
ii := 0;  
  WHILE ii <= NN  
  DO aa(ii) := 0 ; ii := ii + 1  
  INVARIANT ???  
  VARIANT ???  
  END
```

We want to establish that aa is initialised to 0 everywhere: that

$\forall jj. (jj:0..NN \Rightarrow aa(jj) = 0)$

(or equivalently, that $aa=0..NN*\{0\}$.)

What should the INVARIANT and VARIANT be ?

◇ Another example ◇

We want a loop which will find the index ii of an array aa whose corresponding entry is a particular value vv .

The postcondition is $aa(ii) = vv$.

It can only succeed if vv appears somewhere in the array, i.e. $vv \in \text{ran}(aa)$, so this can also appear in the invariant.

```
ii := 0 ;  
  WHILE aa(ii) /= vv  
  DO ii := ii + 1  
  INVARIANT ???          VARIANT ???  
END
```

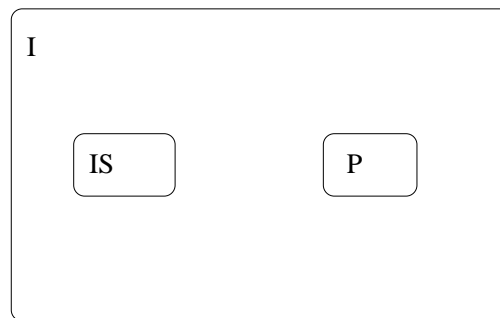
◇ Developing loops ◇

The aim is to write a loop which, starting in a state which satisfies IS , reaches a final state in which P is true.

It should be easy to reach IS .

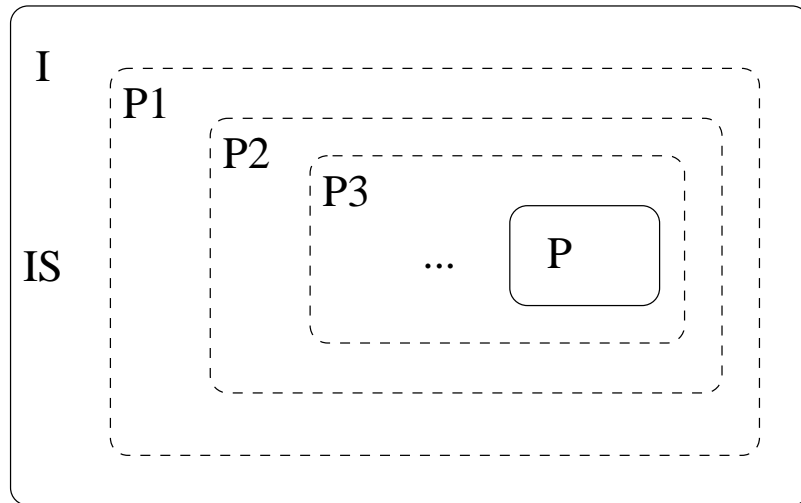
We will try to find an invariant I and variant *before* writing the loop. They can help to develop the loop itself.

The balloon view



An invariant of the loop I must be true before and after termination. The set of states represented by I must contain the set of possible initial states represented by IS , and the set of possible final states represented by P .

◇ Deflating the balloon ◇



P is considered to be the deflated part of the balloon, which is inflated to $P_0 = I$ just before execution of the loop. Each iteration lets some air out of the balloon, until the deflated state P is reached.

The problem is: how do we know how to blow up the balloon?

We generally know what the loop is intended to achieve: P.

So, start from P, and blow it up (ie weaken it) to encompass IS.

◇ Weakening a predicate ◇

There are various ways of weakening a predicate P:

1. Delete a conjunct: e.g. $A \ \& \ B \ \& \ C$ becomes $A \ \& \ C$.

2. Replace a constant by a variable: e.g.

$$\text{sum} = \sum_{i=0}^n a(i)$$

can have the n replaced by j to produce

$$(\text{sum} = \sum_{i=0}^j a(i)) \ \& \ (0 \leq j \leq n)$$

The aim is to weaken the postcondition P to an invariant I.

◇ Deleting a conjunct ◇

Example: want to approximate the square root of n .

Postcondition: a should be the largest integer that is at most \sqrt{n}

$P: 0 \leq a^2 \leq n < (a+1)^2$

Rewriting as a set of conjuncts:

$P: 0 \leq a^2 \ \& \ a^2 \leq n \ \& \ n < (a+1)^2$

Deleting the third conjunct yields a possible invariant: $I: 0 \leq a^2 \leq n$

For the guard of the loop, use the *negation* of the deleted conjunct:

$E: \text{not}(n < (a+1)^2)$

because then the second loop condition holds automatically: $(I \ \& \ \text{not } E) \Rightarrow P$

◇ Loop body ◇

We have so far developed the structure:

```
WHILE  $(a+1)^2 \leq n$   
DO S  
INVARIANT  $0 \leq a^2 \leq n$   
VARIANT v  
END
```

though we don't yet know S or v.

We need to find S such that the first loop condition holds: $(I \ \& \ E) \Rightarrow [S]I$ i.e.

$$\begin{aligned} (0 \leq a^2 \leq n \ \& \ (a+1)^2 \leq n) \\ \Rightarrow [S] (0 \leq a^2 \leq n) \end{aligned}$$

Question: can you find a suitable S ?

[HINT: we know that $(a+1)^2 \leq n$ before executing S, and we want $a^2 \leq n$ after executing S.]

◇ Finishing up ◇

Question: for this S, can you find a suitable variant function? (i.e. one which makes conditions 3 and 4 true).

Question: the loop should start in a state in which the invariant is true. Is there an easy way of initialising a to make the invariant true at the start of the loop (via an 'initialisation' step T)?

The whole developed program is then

T ; LOOP, and

$n \geq 0 \Rightarrow [T; \text{LOOP}]P$

◇ Strategy ◇

When deleting a conjunct from R to produce an invariant P , try using the complement of the deleted conjunct for the guard E of the loop.

Delete a conjunct which makes the remaining predicate easy to establish (via an 'initialisation' step). This means there should be some 'obvious' initial values which make it true.

◇ Replacing a constant ◇

Example: want to sum an array:

Replace a constant by a variable: e.g.

$$P: \text{sum} = \sum_{i=0}^n a(i)$$

can have the n replaced by j to produce

$$I: \text{sum} = \sum_{i=0}^j a(i) \ \& \ 0 \leq j \leq n$$

We can think of sum and j as being variables local to the loop; under the control of the loop.

The final value of the variable j should be the constant n which was replaced, so that: $I \ \& \ (j = n) \Rightarrow P$ The negation

$$(j \neq n)$$

can therefore be the guard of the loop, so that the second loop condition holds automatically:

$$I \ \& \ \text{not}(j \neq n) \Rightarrow P$$

◇ Loop body ◇

We have so far developed the structure:

WHILE $j \neq n$

DO S

INVARIANT $\text{sum} = \sum_{i=0}^j a(i) \ \& \ 0 \leq j \leq n$

VARIANT v

END

though we don't yet know S or v .

We need to find S such that the first loop condition holds:

$(I \ \& \ E) \Rightarrow [S] I$

i.e.

$$\begin{aligned} \text{sum} &= \sum_{i=0}^j a(i) \ \& \ 0 \leq j \leq n \ \& \ j \neq n \\ &\Rightarrow [S] (\text{sum} = \sum_{i=0}^j a(i) \ \& \ 0 \leq j \leq n) \end{aligned}$$

◇ Finding S ◇

Question: can you find a suitable S ? The trick is to find one which is closer to termination.

Observation: The fact that $j < n$ before S is executed will allow it to be incremented while remaining between 0 and n , provided the rest of the invariant can also be made consistent by S.

Question: for this S, can you find a suitable variant function? (i.e. one which makes conditions 3 and 4 true).

Question: the loop should start in a state in which the invariant is true. Is there an easy way of initialising `sum` and `j` which make the invariant true at the start of the loop (via an 'initialisation' step T)?

The whole developed program is then T ; LOOP,
and $n \geq 0 \Rightarrow [T; LOOP]P$

◇ Another example ◇

We want a program which should calculate 2^K and put the result in `mm`. i.e. its specification is `mm := 2^K`.

Replacing `K` by `nn` yields the potential invariant:

$$mm = 2^{nn}$$

and with a guard of `nn /= K` we can derive the loop

```
mm, nn := 1, 0 ;
  WHILE nn /= K
  DO mm, nn := mm*2, nn+1
  INVARIANT mm = 2^nn
           & nn:NAT & nn<=K
  VARIANT K - nn
END
```

◇ Loop pragmatics in B ◇

- ◇ Loops can appear only in implementation machines, so there will not be any state of their host machine to manipulate. Instead, they will manipulate the state of imported machines via their operations. They will also manipulate local variables, if the loop is inside the scope of a VAR.

The invariant of a loop will therefore be concerned with the variables of imported machines, and its local variables.

- ◇ Often, extra clauses concerning the types of variables have to be brought in to the loop invariant in order to establish the conditions concerning the variant. For example, on the previous slide, the extra clauses ' $nn : \text{NAT} \ \& \ nn \leq K$ ' were required to establish that $K - nn$ is a suitable variant.

◇ Composing Specifications ◇

There are two principal ways in which one machine can use another: one way is to use it as a slave machine, being able to change its state. The other is to use it to answer queries and perform functions, but without being able to change its state.

- ◇ EXTENDING machines
- ◇ INCLUDING machines
- ◇ PROMOTING operations
- ◇ IMPORTING machines
- ◇ USING machines
- ◇ SEEING machines
- ◇ Development

◇ Structure ◇

Modularity

- ◇ Separation of concerns (into different state components)
- ◇ Separation of proof obligations
- ◇ Aids understanding
- ◇ Allows module reuse

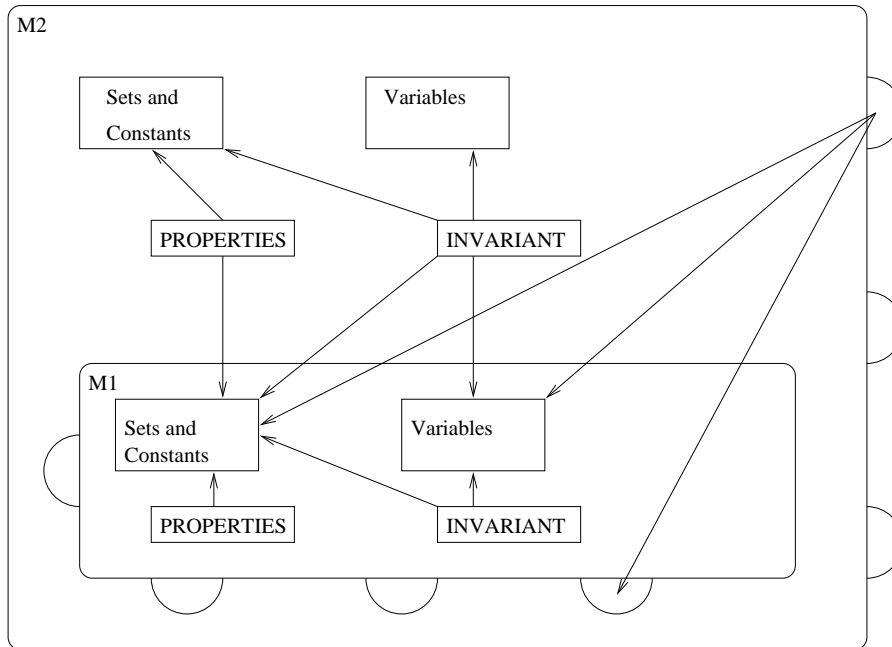
Abstraction

- ◇ Information hiding
- ◇ Separation of specification from implementation concerns
- ◇ Reasoning at appropriate level

Modularity and abstraction are the principal weapons in the battle against complexity.

◇ INCLUDES ◇

M2 INCLUDES M1:



Update access: only operations of M1 can update variables of M1; operations of M2 can update variables of M2 only

Difference to IMPORTS: M2 can read variables of M1

◇ INCLUDES ◇

The machine $M1$ is a machine defined independently of any machines that INCLUDE it, and so it cannot refer to any of the information contained in $M2$.

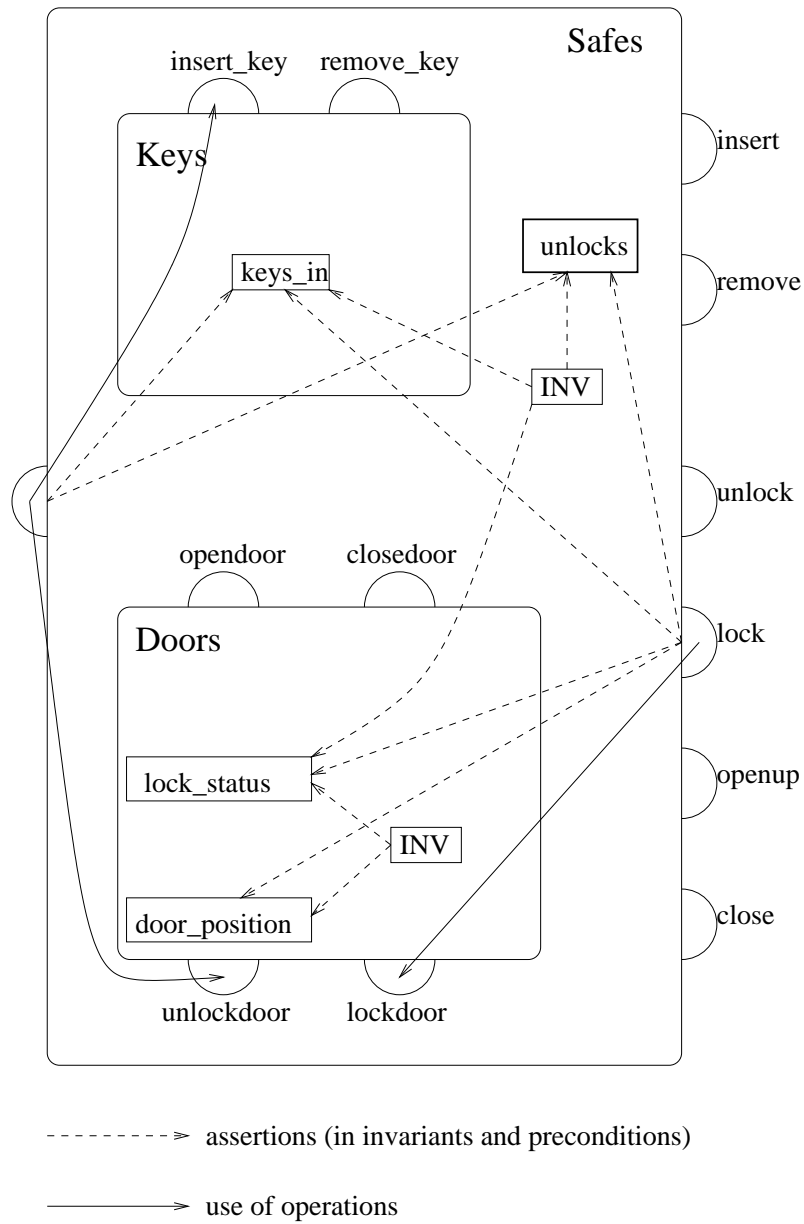
The invariant and operations of the machine $M2$ have access to the state and operations of $M1$. $M2$ can change the state of $M1$ only through $M1$'s own operations.

This allows a separation of proof obligations and reuse of proofs. No further proofs need be carried out on $M1$ when it is included in another machine.

Example:

```
MACHINE Safes  
INCLUDES Keys, Doors
```

◇ Structure of Safes ◇



◇ Inheritance ◇

An abstract machines may be structured by use of an **INCLUDES** clause, which lists a number of abstract machines (and supplies their parameters) which are incorporated into the **EXTENDING** machine.

In $M2$ **INCLUDES** $M1$:

- ◇ The *native* variables of $M2$ are those defined in $M2$'s **VARIABLES** clause.
- ◇ Similarly, the *native* sets and constants are those defines in $M2$'s **SETS** and **CONSTANTS** clauses.
- ◇ The *included* variables of $M2$ are the native and included variables of $M1$.
- ◇ Similarly for sets and constants

◇ PROMOTES ◇

In some situations it is desirable to *promote* some operations of an INCLUDED machine to become operations of the INCLUDING machine, but not to promote all of them.

The **PROMOTES** clause lists all of those operations which should be promoted to the interface of the resulting machine.

Since these operations are now operations of the INCLUDING machine, they must be shown to preserve that machine's invariant:

The proof obligation

$$I1 \wedge I2 \wedge P \Rightarrow [S](I1 \wedge I2)$$

must be proven for each promoted operation
PRE P THEN S END.

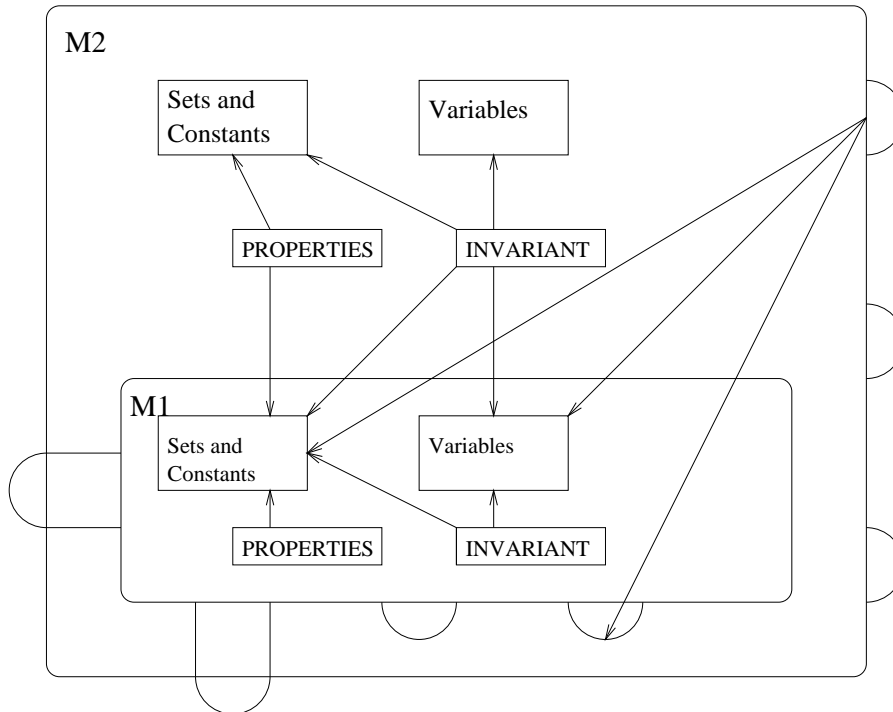
MACHINE Safes

INCLUDES Keys, Doors

PROMOTES open_door, close_door

◇ PROMOTES ◇

M2 PROMOTES M1:



◇ EXTENDS ◇

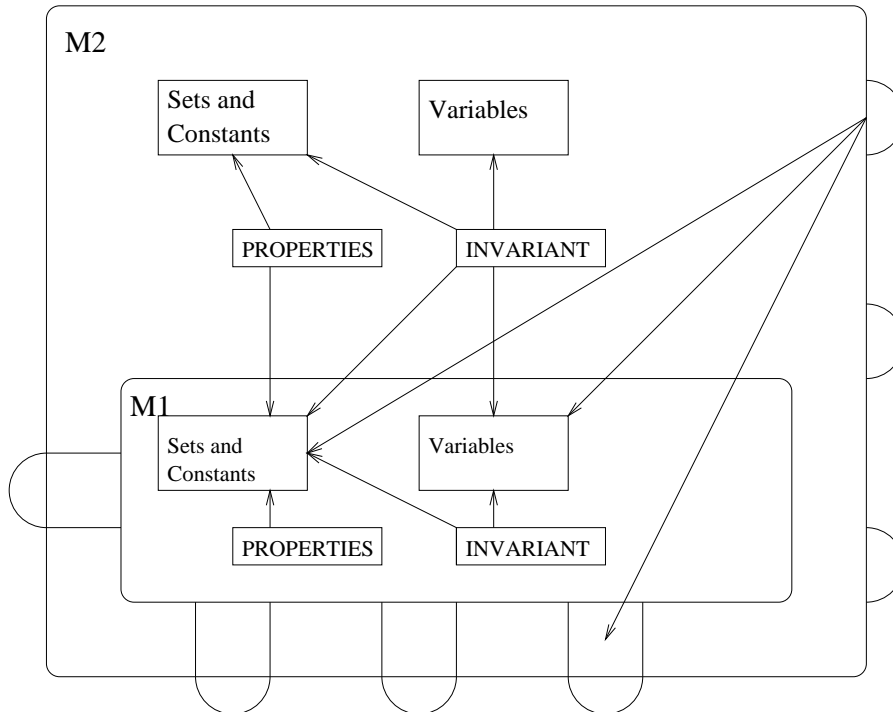
EXTENDS is a special case of INCLUDES.

In $M2$ EXTENDS $M1$:

- ◇ All operations of $M1$ are promoted to operations of $M2$
- ◇ All included variables of $M2$ may be directly read-accessed in operation definitions of $M2$.
- ◇ Included variables of $M2$ may be updated in $M2$ only by use of their native machine's operations (which may appear in $M2$'s operation definitions).
- ◇ The invariants of $M1$ becomes part of the invariant of $M2$.
- ◇ Any clause of $M2$'s invariant (inherited or defined explicitly in $M2$ may refer to included variables.

◇ EXTENDS ◇

M2 EXTENDS M1:



All of the operations of $M1$ are promoted to operations of $M2$.

◇ Restrictions ◇

A machine may be INCLUDED in at most one other machine. This prevents machines from violating each other's invariant.

INCLUDES is a *semi-hiding* operator. It offers some modularity, but not full abstraction.

◇ Verifying operations ◇

Operations of INCLUDED machines may be used in operation definitions of the INCLUDING machine.

When verifying these, the operation must be replaced by its definition, with the actual parameters substituting the formal ones.

Example:

```
openup(dd) =  
PRE dd : DOOR & unlocks (dd) : keys_in  
& lock_status(dd) = unlocked &  
door_position(dd) = closed  
THEN opendoor(dd)  
END
```

uses the following operation from *Doors*:

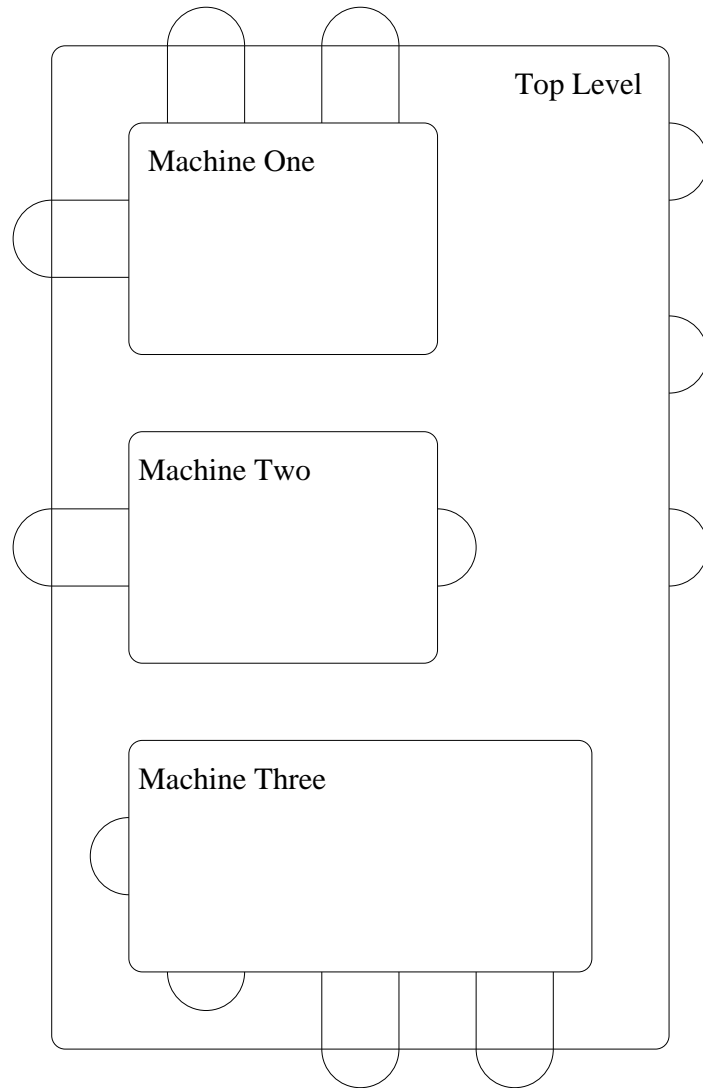
```
opendoor(dd) =  
PRE dd : DOOR & lock_status(dd) = unlocked  
THEN door_position(dd) := open  
END
```

◇ Preconditions ◇

The precondition of an INCLUDEd operation is included in its expansion.

Hence it will have to be true whenever the operation is called in order for consistency to be obtained. Hence the INCLUDING machine must guarantee that the precondition is true whenever the operation is called.

◇ Summary ◇



◇ Summary ◇

MACHINE

Top_Level

EXTENDS

Machine_One

INCLUDES

Machine_Two, Machine_Three

PROMOTES

Op_Two_1, Op_Three_3, Op_Three_4

VARIABLES

var_1, var_2

INVARIANT

etc

OPERATIONS

etc

END

Then no other machine may EXTEND or INCLUDE Machines One, Two or Three. However, *Top_Level* may be included in some other machine.

◇ USES ◇

$M2$ USES $M1$ means that $M2$ has read-access to the variables of $M1$, and may invoke query operations of $M1$: those operations that do not change its state. (This may be statically checked.)

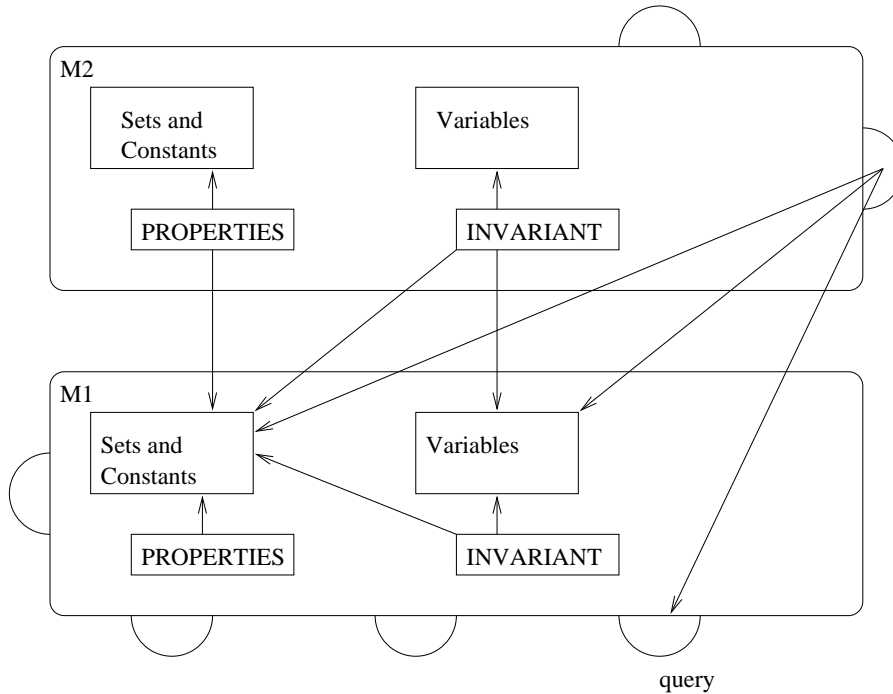
$M2$ may also refer to the state variables of $M1$ in its INVARIANT. This means that even though $M2$ does not have control over the state of $M1$ it still requires a particular relationship between the variables of the two machines.

$M1$'s variables become *used variables* of $M2$. This means they may appear in the invariant of $M2$ and on the right hand side of assignments in initialisation and operation definitions.

A machine may appear in a number of USES clauses of other machines.

◇ USES ◇

M2 USES M1:



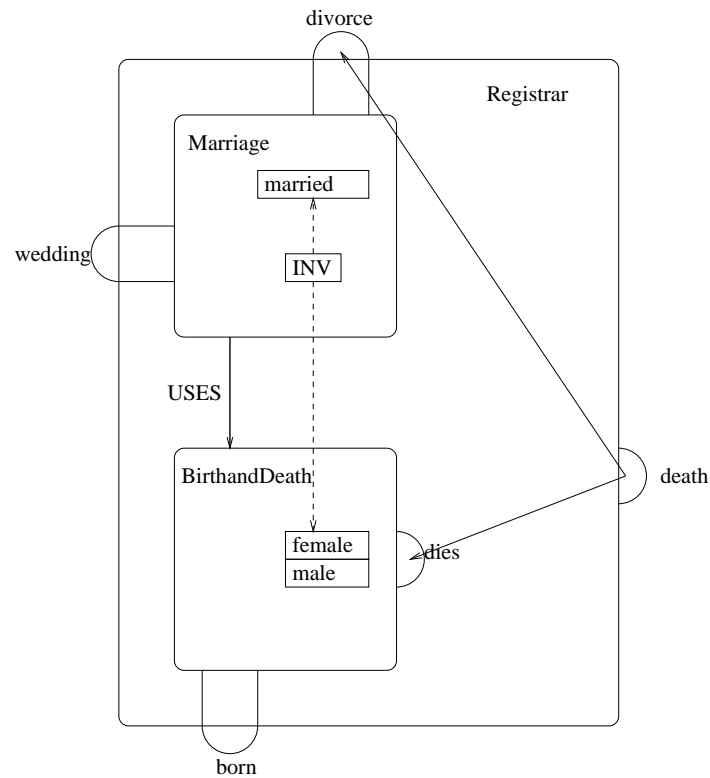
Example:

Marriage USES BirthandDeath

Note that the INVARIANT of $M2$ is concerned with the variables of $M1$ even though $M2$ has no control over the operations of $M1$, and therefore no control over how those variables change.

◇ A Registrar ◇

(see attached handout)



- ◇ Invariant of Registrar?
- ◇ Consistency of each machine?

Add a sexchange operation to BirthandDeath.

◇ Consistency ◇

If $M2$ USES $M1$ then for $M2$ to be consistent it must be shown that all of the operations of $M2$ preserve its invariant. Since the invariant of $M2$ will include a linking invariant concerning the relationship between its variables and the variables of $M1$, this means it is necessary to prove that the linking invariant is not violated by $M2$'s operations.

However, it is not necessary to check that the operations of $M1$ preserve $M2$'s invariant in order to prove consistency of $M2$. In general, this might not be true.

This proof obligation will arise for a machine which includes both $M1$ and $M2$: $M2$'s invariant and $M1$'s operations will both be part of this machine, and this is the level at which their consistency needs to be checked.

◇ SEES ◇

$M2$ SEES $M1$ means that $M2$ has read-access to the variables of $M1$, and may invoke query operations of $M1$. The only difference between SEES and USES is that the variables of $M1$ cannot appear in the invariant of $M2$. This means that there are no parts of the invariant which need to be passed up the hierarchy of abstract machines.

$M2$ may be an abstract machine, a refinement or an implementation. If $M2$ is an abstract machine or a refinement, then $M1$'s variables may be read directly in initialisation and operation definitions.

If $M2$ is an implementation then $M1$'s variables are not available to $M2$. Only the query operations of $M1$ may appear in the definition of $M2$. This supports *full hiding*.

A machine may appear in a number of SEES clauses of other machines.

◇ SEES ◇

SEES is often used to allow many machines within a development to refer to a particular component.

If a MACHINE SEES another MACHINE, it is common for its refinements also to see that machine.

This permits structuring of the development, and separation of concerns.

If $M2$ SEES $M1$ (rather than USES) then there are no proof obligations carried up from $M2$ to any INCLUDING machines. Hence it is preferable to make use of SEES rather than USES where possible.